# NAVAL POSTGRADUATE SCHOOL
# Monterey, California

# THESIS

W35 1

DESIGN AND IMPLEMENTATION OF A PRETTY PRINTER
FOR THE
FUNCTIONAL SPECIFICATION LANGUAGE SPEC

by

Jill A. Weigand

June 1988

Thesis Advisor: Valdis Berzins

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

| | | | |
|---|---|---|---|
| REPORT SECURITY CLASSIFICATION UNCLASSIFIED | | 1b RESTRICTIVE MARKINGS | |
| SECURITY CLASSIFICATION AUTHORITY | | 3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited | |
| DECLASSIFICATION / DOWNGRADING SCHEDULE | | | |
| PERFORMING ORGANIZATION REPORT NUMBER(S) | | 5 MONITORING ORGANIZATION REPORT NUMBER(S) | |
| NAME OF PERFORMING ORGANIZATION Naval Postgraduate School | 6b OFFICE SYMBOL (If applicable) Code 37 | 7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School | |
| ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000 | | 7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000 | |
| NAME OF FUNDING / SPONSORING ORGANIZATION | 8b OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER | |
| ADDRESS (City, State, and ZIP Code) | | 10 SOURCE OF FUNDING NUMBERS | |

| PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO | WORK UNIT ACCESSION NO. |
|---|---|---|---|
| | | | |

TITLE (Include Security Classification)
DESIGN AND IMPLEMENTATION OF A PRETTY PRINTER FOR THE FUNCTIONAL SPECIFICATION LANGUAGE SPEC

PERSONAL AUTHOR(S)
Weigand, Jill A.

| TYPE OF REPORT | 13b TIME COVERED | 14 DATE OF REPORT (Year, Month, Day) | 15 PAGE COUNT |
|---|---|---|---|
| Master's Thesis | FROM ____ TO ____ | 1988 June | 235 |

SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

| COSATI CODES | | | 18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | Pretty printer; attribute grammar; Functional specification |
| | | | |
| | | | |

ABSTRACT (Continue on reverse if necessary and identify by block number)

The purpose of this thesis is to develop and implement a language dependent pretty printer for the SPEC language. SPEC is a formal language for writing black-box specifications for components of software systems which are developed in the functional specification stage of software development. The pretty printer is a software tool used to format specifications to make them easier to understand and read. A computer program was written implementing the pretty printer design criteria. The program uses Kodiyak and was written as an attribute grammar. Included is a listing of the grammar for the SPEC language, the pretty printer program source listing, a representative sample of input used to test the pretty printer program and resulting output. A significant result of this study is the conclusion that by abstracting this language dependent pretty printer it is feasible to use Kodiyak to create a language independent pretty printer generator.

| | | |
|---|---|---|
| DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED | |
| 2a NAME OF RESPONSIBLE INDIVIDUAL Prof. Valdis Berzins | 22b TELEPHONE (Include Area Code) (408) 646-2461 | 22c OFFICE SYMBOL 52Be |

DD FORM 1473, 84 MAR    83 APR edition may be used until exhausted. All other editions are obsolete    SECURITY CLASSIFICATION OF THIS PAGE

# Design and Implementation of a Pretty Printer for the Functional Specification Language SPEC

by

Jill Annette Weigand
Lieutenant, United States Navy
B.S., United States Naval Academy, 1981

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June, 1988

## ABSTRACT

The purpose of this thesis is to develop and implement a language dependent pretty printer for the SPEC language. SPEC is a formal language for writing black-box specifications for components of software systems which are developed in the functional specification stage of software development. The pretty printer is a software tool used to format specifications to make them easier to understand and read. A computer program was written implementing the pretty printer design criteria. The program uses Kodiyak and was written as an attribute grammar. Included is a listing of the grammar for the SPEC language, the pretty printer program source listing, a representative sample of input used to test the pretty printer program and resulting output. A significant result of this study is the conclusion that by abstracting this language dependent pretty printer it is feasible to use Kodiyak to create a language independent pretty printer generator.

TABLE OF CONTENTS

## LIST OF FIGURES

# I. INTRODUCTION

An expanding field of interest in computer science is that of software engineering. It is important to the development of software systems that critical work be done early in the design of software systems. Therefore the development of a functional specification is of great importance to the design effort. Berzins [Ref. 1] developed a formal language SPEC for writing black-box specifications for components of software systems in the functional specification stage of software development. To increase the readability of this code a software tool was designed to format SPEC. This thesis centers on the design and implementation of a language dependent pretty printer for the SPEC language.

## A. OBJECTIVES

The main objective of this thesis is to design and implement a language dependent pretty printer to format SPEC. Appendix A [Ref. 2] contains a listing of the grammar for SPEC language. The code for the pretty printer is written as an attribute grammar. This code is then compiled using Kodiyak. The output of the Kodiyak compiler is the executable code for the pretty printer to format SPEC.

Additionally, it is desired that the pretty printer code be easy to read and change. Since this is a research project code efficiency is not as important as code readability. It is essential that the pretty printer code operate correctly and that other researchers can understand and modify the code.

The final objective is to try and take what is learned in this implementation, abstract it, and develop guidelines for a language independent pretty printer using Kodiyak and attribute grammars. These general guidelines are a direct result of the insight gained from the design and implementation of one pretty printer using Kodiyak and attribute grammar.

## B. RESEARCH QUESTIONS

There are two research questions for this thesis. First, what are the underlying issues and decisions that must be made in the design and implementation of a language dependent pretty printer? Once these issues are identified what is their impact on readability, ease in modifying, portability and efficiency?

Second, can a language independent pretty printer be developed from the methodology used in this thesis in the development of the language dependent pretty printer? Is the implementation regular enough to abstract general rules and guidelines or is the implementation based on many special cases and no general rules and guidelines are possible?

## C. SUMMARY OF FINDINGS

The final output from the research conducted in this thesis is the implementation of a working language dependent pretty printer. Though it may not be optimally efficient it works correctly and the code is readable. Appendix B contains the pretty printer code listing and Appendix C [Ref. 3] contains sample input and output for the pretty printer.

Additionally, from the work done in the development of the pretty printer a set of general guidelines is developed for the design of a language independent pretty printer generator. This generator will need a preprocessor. Output of the preprocessor is transmitted to the Kodiyak compiler with the output from the compiler an executable code for the pretty printing of the language defined by the input to the preprocessor.

## D. ORGANIZATION OF STUDY

Chapter II lays the theoretical framework for the design and implementation of the pretty printer. The details of attribute grammars, Kodiyak, pretty printers and SPEC are discussed. These four areas are heavily used in Chapter III and Chapter IV. Readers unfamiliar with these subjects will find this chapter necessary.

Chapter III explains the design questions and design decisions that are made in this implementation. Additionally an explanation of the attributes used to create the pretty printer are outlined. The final subject in this chapter is the general and specific rules for the pretty printer. A user of the pretty printer code needs only a working knowledge of the general rules. A person who is going to modify the pretty printer or wants more details will find the specific rules define the operation of the pretty printer.

Chapter IV covers the conclusions of this thesis. The conclusions address the issues of the specific implementation and how this implementation can be extended to generate a language independent pretty printer generator. The lessons learned in the design of the language dependent pretty printer are abstracted to produce guidelines for the future development of a language independent pretty printer generator using Kodiyak and attribute grammar.

# II. BACKGROUND THEORY

This chapter lays the theoretical framework for the design and implementation of the pretty printer. In this chapter details of the following four subject areas will be discussed:

(1) Attribute grammars
(2) Kodiyak
(3) Pretty printer
(4) SPEC

Chapter III and Chapter IV of this thesis draw heavily on developments in these four areas. Readers unfamiliar with these subjects will find this chapter necessary.

## A. ATTRIBUTE GRAMMARS

A grammar consists of a finite set of nonterminal symbols, a finite set of terminal symbols, a set of production rules and a start symbol. A language defined by a grammar is a set of strings consisting only of terminal symbols that can be generated by the grammar. [Ref. 4:p. 81] A nonterminal symbol is defined as a variable or a syntactic category [Ref. 5:p. 77]. A terminal symbol is defined as a symbol that can appear only on the right-hand side of a production rule [Ref. 6:p. 97] or as a primitive symbol of the language [Ref. 5:p.77].

An attribute grammar is an extension of a context-free grammar whose generated language includes syntax and semantics. A context-free grammar (CFG) is a four tuple G = (N,T,P,S) where N is the set of nonterminals, T is the set of terminals; P is the set of production rules and S is the start symbol [Ref. 4:pp. 84-85]. The syntactic part of the attribute grammar is typically a context-free grammar and the semantic part consists of a set of attributes associated with each symbol and a set of semantic functions used to evaluate the attributes' values in term of the syntactic tree [Ref. 7:p. 331].

The attributes provide a means for expressing data flow in the derivation tree. They are associated with the node in the derivation tree and represent inputs and outputs of the nodes. The attributes of each node in the tree can be defined in terms of the attributes of neighboring nodes.

An important feature of an attribute grammar is that some of the attributes are defined for nonterminals that appear on the right side of the corresponding production rule, while other attributes are defined when the nonterminal appears on the left side of the corresponding production rule. This implies there are two types of attributes. First, a synthesized attribute is based on the attributes of the descendants of the nonterminal symbol. Second, an inherited attribute is based on the attributes of the ancestors. Synthesized attributes are always evaluated from the bottom up in the tree structure, while inherited attributes are always evaluated from the top down in the tree structure. [Ref. 8:p. 130]

Figure 2.1, from Knuth, shows an example of an attribute grammar which gives a precise definition of binary notation for numbers. This binary notation is based on the definition of the context-free grammar (as listed in the left column of Figure 2.1). The terminal symbols are ".", "0" and "1". The nonterminal symbols are "B", "L" and "N" representing respectively bit, list of bits and number. A binary number is intended to be any string of terminal symbols which can be obtained from "N" by application of the production rules listed in the context-free grammar. [Ref. 8:pp. 127-130]

| Syntactic Rules | Semantic Rules |
|---|---|
| B -> 0· | v(B) = 0 |
| B -> 1 | v(B) = 2**s(B) |
| L -> B | v(L) = v(B), s(B) = s(L), l(L) = 1 |
| L1 -> L2B | v(L1) = v(L2) + v(B), s(B) = s(L1)<br>s(L2) = s(L1) + 1, l(L1) = l(L2) + 1 |
| N -> L | v(N) = v(L), s(L) = 0 |
| n -> L1 . L2 | v(N) = v(L1) + v(L2), s(L1) = 0,<br>s(L2) = -l(L2) |

Note: "**" symbolizes exponent
L1, L2 are subscripted

Figure 2.1 Binary Notation Example

The right-hand column of Figure 2.1 represents the semantic rules. The semantic rules define all the attributes of a nonterminal in terms of the attributes of its immediate descendants. Values are ultimately assigned for each attribute. The three attributes used in this example are "v" for value, "l" for length and "s" for scale. Figure 2.2 gives an example of the association between attributes and nonterminals. Figure 2.3 lists the synthesized attributes and the inherited attributes for this example. [Ref. 8:p. 130]

Each B has a "value" v(B) which is a rational number.
Each B has a "scale" s(B) which is an integer.
Each L has a "value" v(L) which is a rational number.
Each L has a "length" l(L) which is an integer.
Each L has a "scale" s(L) which is an integer.
Each N has a "value" v(N) which is a rational number.

Figure 2.2 Nonterminals with Assigned Attributes

| Synthesized Attributes | Inherited Attributes |
|---|---|
| v(B) | s(B) |
| v(L) | s(L) |
| l(L) | |
| v(N) | |

Figure 2.3 Synthesized/Inherited Attributes

The semantic rules may be used to give a "meaning" to strings of the context-free language. For any derivation of a terminal symbol "t" from "S" (the start symbol) by a sequence of production rules the derivation tree is constructed. Then the attributes of this derivation tree, or parse tree, are evaluated. This process of attribute definition, which can occur in any order, is applied throughout the tree until no more attribute values can be defined. The defined attributes at the root of the tree constructed give the "meaning" (or desired answer or output) of that derivation tree. [Ref. 8:pp. 132-133]

Any attribute grammar can be composed of both synthesized and inherited attributes. Semantic rules that do not use inherited attributes can be considerably more complicated (along with being harder to understand and to manipulate) than semantic rules which allow both kinds of attributes. Synthesized attributes alone are sufficient to define any function of the derivation tree but, in practice, using both types of attributes leads to simplifications. [Ref. 8:p. 134]

Appendix A [Ref. 2] lists the context-free grammar used in SPEC. No attributes are listed in this grammar. Appendix B lists the entire language Spec including the attributes for the pretty printer.

## B. KODIYAK

The Kodiyak compiler is essential to the implementation of this pretty printer. A complete understanding of Kodiyak is not necessary for the user of the pretty printer but an implementor will be interested in the details. The following is a summary of the major points of Kodiyak with an emphasis on what is used in the actual code of the pretty printer. It covers the following topics:

(1) Description of Kodiyak
(2) Format
(3) Output procedures
(4) Using Kodiyak

All of the points covered in the following section come directly from The Incompleat AG User's Guide and Reference Manual [Ref. 9:pp. 2-25]. The following is a synopsis of the major points of this manual, enough to understand the code for the pretty printer but not all the details. If further or more detailed information is needed consult the user's guide mentioned above.

### 1. Description of Kodiyak

Kodiyak is a language designed for constructing translators. It is modeled after Knuth's description of attribute grammars. The Kodiyak translator accepts a context-free grammar along with attribute declarations and equations, a scanner specification, and output declarations, and generates an executable translator.

### 2. Format

The Kodiyak program is divided into three sections. The first section describes the features of the lexical scanner. The second section names the attributes associated with each grammar symbol and declares their types. The third and final section describes the grammar and attribute equations which define the semantics of the translation. The sections are separated by the unique symbol double percent symbol ("%%") which is located on a line by itself.

a. Lexical Scanner Section

Each statement in the first section of the Kodiyak program describes the terminal symbols of the translation in some way. The two functions of this section are first, to specify the correspondence between the terminal symbols of the grammar and the input text, and second, to specify a set of operator precedences to be used in conjunction with the grammar.

The lexical scanner section defines the set of substitutions to be performed on the input text. These substitutions are carried out using regular expressions. Input is scanned for text that matches these regular expressions. If a match is found then the corresponding text is deleted and replaced with the associated named terminal symbol (the symbol that occurs on the left-hand side of the regular expression).

Figure 2.4 shows examples of regular expressions. An explanation of the symbols will assist in understanding this terminology. The colon (":") indicates that a regular expression follows. The bar ("|") indicates an OR. The symbol "Backslash" is a symbolic constant representing the "\" character. The input string "MOD" or the symbol "Backslash" is replaced by the atomic terminal symbol "MOD". The asterisk ("*") is an indication of zero or more repetitions. This means that the symbol {Blank}* represents zero or more instances of the symbolic constant "Blank". The plus sign ("+") is very similar to the asterisk except that it means one or more. The square brackets ("[", "]") mean an OR operation of the items inside of the brackets. Therefore [a-zA-Z] means one letter of the alphabet. The curly brackets ("{","}") are used to invoke substitution. The caret ("^") is used to mean everything except what follows it. Therefore [^"\\] means everything except the quotation mark or a backslash.

```
COMMENT    : "--".*"\n"
AND        :"&"
MOD        :{Backslash}|MOD
           :{Blank}+
NAME       :[Letter}{alpha}*


%define Char   :{[^"\\]|{Backslash}{Quote}}
%define Letter : [a-zA-Z]
%define Int    : {Digit}+
%define Digit  : [0-9]
%define Quote  : ["]
```

Figure 2.4 Regular Expressions

Another way to write regular expressions is also illustrated in Figure 2.4. It is a shorthand or abbreviated way to write a regular expression. This second method uses "%define" as the first symbol in the regular expression. Either format can be used with regular expressions and they both can appear in the code at the same time. There are three important rules about regular expressions that must be remembered to prevent errors. First regular expressions can occur on one line only. They may never extend beyond one typed line. Second, regular expression substitution may not be used outside of the regular expression section of the lexical scanner. Third, each regular expression must be defined before it can be used.

#### b. Attribute Declaration Section

The attribute declaration section of the Kodiyak program names the attributes associated with each grammar symbol and declares their types. In this version of the Kodiyak program the attribute declarations for all nonterminals and named terminals are the only statements that may be present in this section. Future versions of Kodiyak may include other features.

Kodiyak supports two primitive data types for the attributes. They are integers and strings. All simple mathematical functions are available for use with the integers (i.e., addition, subtraction, multiplication, division). The size (minimum or maximum value) of the integers will be machine dependent. Strings can be of any length with the only associated function being concatenation. String constants are denoted by enclosing the string in double quotes. Any special characters must be preceded by the "\" symbol. Figure 2.5 shows an example of integers and strings and how they are declared.

```
renames {
          indent : string;
          str_value : string;
          bcursor : integer;
          padding : string;
          ecursor : integer;
         }
```

Figure 2.5 Integer and String Declaration Example

Kodiyak also supports higher order types called maps which can be used as symbol tables. These will not be discussed here since they are not used by this implementation of the pretty printer.

Named terminal symbols are permitted to have user-defined attributes as well as two special predefined attributes %text and %line. They are initialized to the text of the terminal symbol matched and the line number of the input text that the match text occurred on (respectively). The attribute %text is used in the pretty printer but the attribute %line is not.

### c. Grammar and Attribute Equation Section

The third and final section of the Kodiyak program describes the grammar and attribute equations which define the syntax and the semantics of the translation. The left-hand symbol of the first rule of the grammar section is taken to be the start symbol. The start symbol may not appear on the right hand side of any rule.

An attribute value is named by naming the grammar symbol associated with it, a period and the attribute. The grammar symbol may be named in one of three ways. The simplest (if there are no repeated grammar symbols in the production) is use the name of the symbol. If that grammar symbol appears more than once then further distinction must be made. In this case, the name is taken to refer to the left-most occurrence of the grammar symbol. To name a grammar symbol which is not the left-most instance of that grammar symbol, the name may be indexed by a number in square brackets denoting which occurrence is desired. The left-most occurrence of the symbol is numbered one, the next left-most two, the next left-most three, etc. These first two ways are used by the pretty printer. The third way is with the use of the dollar sign ("$") followed by the numerical position of the grammar symbol in the production. This is not used by the pretty printer. Figure 2.6 gives examples of naming attributes.

All of the functions/operators available for the integer and strings are shown in Figure 2.7. These functions include arithmetic, string manipulation, relational operators and logical operators. The left-hand column of Figure 2.7 shows the symbol and the right-hand column provides the meaning of that symbol.

```
expr:
       expr '+' UNUMBER
       {
          $$.value = $1.value + s2i($3.%text);
       }
expr:
       expr '+' UNUMBER
       {
         expr.value = expr[2].value + s2i(UNUMBER.%text);
        }
expr:
       expr '+' UNUMBER
       {
          expr[1].value = expr[2].value +
                              s2i(UNUMBER[1].%text);
       }
```

Figure 2.6 Naming Attribute Examples

| | |
|---|---|
| + | addition |
| * | multiplication |
| - | subtraction |
| / | division |
| ^ | concatenation |
| [ ] | concatenation |
| < | less than |
| > | greater than |
| == | equal |
| <> | not equal |
| <= | less than or equal |
| >= | greater than or equal |
| && | and |

Note this a partial list showing just the operators used in this implementation.

Figure 2.7 Available Operators

One final note about attribute equations concerns the if-then-else statement. The symbology is a little different from more programming languages. There are no keywords "if", "then" or "else" but only the symbols "->" and "#". For example, "IF A THEN B ELSE C" is written "A -> B # C". Figure 2.8 shows a typical if-then-else statement. The expression to the right of the "=" is the "if". The expression to the right of the "->" is the "then" and the expression to the right of the "#" is either the "else" or "else_if" part.

```
IF THEN ELSE EXAMPLE

     comment[1].str_value = comment[1].bcursor <= 0
       -> [COMMENT.%text,comment[2].str_value]
       #  ["\n",COMMENT.%text,comment[2].str_value];


IF THEN ELSE_IF EXAMPLE

     comment[1].str_value = comment[1].bcursor <= 0
       -> [COMMENT.%text,comment[2].str_value]
       # comment[1].bcursor + len(COMMENT.%text) >= 80
       -> ["c",COMMENT.%text,comment[2].str_value]
       #  {"\n",COMMENT.%text,comment[2].str_value];
```

Figure 2.8 If Then Else Statement

### 3. Output Procedures

To get any output from the Kodiyak program, such as the pretty printer does, the program must include a special side-effecting procedure. The side-effecting procedure with its associated equations can only be used by the start symbol. There are five of these procedures available in Kodiyak and they are introduced by a percent symbol, followed by the name of the procedure and the arguments surrounded by parentheses. The pretty printer only uses one of the procedures, named "%output". Figure 2.9 gives a short description of the procedure as well as an example.

```
          %output(val:string)

     val is written to the standard output

               EXAMPLE

         start
             :  comment spec
                {
                  %output([comment.str_value,
                           spec.str_value]);
                  comment.bcursor = 0;
                 }
```

Figure 2.9 Output Procedure Example

## 4. Using Kodiyak

The Kodiyak compiler creates and processes many files among them are files which are processed by Yacc, by Lex and by the C compiler. There are also two predefined files that the Kodiyak compilation depends on. The first is the Kodiyak library which contains functions for creating the parse tree, evaluating attributes, concatenating strings, creating pairs, etc. This file is usually not modified by the user. In this implementation this file is named kmain.c.

The second file is the library. This file is a set of C functions. This is where the user can add any new functions for the Kodiyak's use. In this implementation the function SPACES was added to the end of the file named kclib.c. The file itself has information on how to add functions. Note this is the only place available for the user to add his own functions. Creating a separate file and compiling it separately will cause syntax errors.

The command to invoke the Kodiyak is "k sample.k". The file that handles everything (i.e., the driver of the software) is the file k. The name of the file to be compiled is sample.k in this case. Files to be compiled should have the extension ".k" or the compiler may not accept it. Kodiyak programs may also have one other extension ".m4" but that was not used in this implementation.

If the program compiles without errors, the resulting object file (executable code) will be in the file named "sample". This is the name of the file submitted without the extension. Otherwise some cryptic error messages may be printed. Some common errors (as well as the type of error message printed and "what to do") are listed in Figure 2.10. There is also a set of options available to assist in debugging. The options are typed on the same line but immediately after the "k sample.k". Figure 2.11 shows the options available.

```
ERROR

    1. syntax error  - which are mostly typing errors
    2. missing attribute evaluation rules and
       circular evaluation rules
    3. table overflow errors
    4. memory storage exceeded

MESSAGE PRINTED OUT

    1. an associated line number with the symbol
       that has the syntax error
    2. pair of grammar rules  the parent of the error
       rule and the error rule
    3. statement "table overflow need more space"
    4. statement "exceeded memory space"

WHAT TO DO

    1.  find the typing mistake
    2.  look for a missing attribute evaluation rule in
        the parent rule
    3.  increase table size allotted
    4.  increase memory space for given variable
```

Figure 2.10 Common Errors in Kodiyak

One final note about error messages. The first character of any identifier name should usually be ignored. They are tacked on by the Kodiyak compiler to avoid naming conflicts between Lex, Yacc, C and Kodiyak library routines. As an example, if the error message printed "ucomment.padding is undefined" this should be read as "comment.padding is undefined".

|  |  |
|---|---|
| -h | Print out a list of legal options. |
| file | Read input from "file"rather than the standard input |
| -e | Continue attribute evaluation even after an error occurs. This is useful when debugging attribute definitions. |
| -l | Print out all tokens as they are scanned. |
| -y | Print out all grammar rule reductions as they occur. |
| -L | Turn on LEX's debugging features. |
| -Y | Turn on YACC's debugging features. |
| -c | Generate a core image when a run-time error occurs |
| -s | Print out storage statistics after all attribute evaluations is completed. |
| -o file | Divert the standard output to "file". |

Figure 2.11 Options List

## C. PRETTY PRINTER

A pretty printer is a software tool used to format programs to make them easier to understand and read. It takes character strings, called tokens, from an input source and prints them with aesthetically appropriate spacing and line breaks. The input is usually a text file or a parse tree. The two primary functions of the pretty printer are to insert spaces and linefeeds between tokens and to determine where and how to break lines. [Ref. 10:p. 119]

A syntax-directed pretty printer is a pretty printer that knows the syntax of a programming language and formats programs based on that knowledge. The syntactic structure and flow of control of pretty printed programs are made clear because the output medium shows the indentation and line breaks. [Ref. 10:p. 119]

A syntax-directed pretty printer may be either language dependent or language independent. A language dependent syntax-directed pretty printer is written for a specific language. Since all constructs in the language are known, the pretty printer traditionally has been written as a large switch or case statement (for languages like Pascal and Lisp). If any changes are made to the language the code for the pretty printer must be revised. [Ref. 10:p. 120]

The language independent syntax-directed pretty printer is designed to be used for any language. In this case no knowledge of any particulars of a language are used in constructing the pretty printer. This version of a pretty printer must be given information about a language to produce a pretty printer for that language. In the long run it is easier and quicker to write one language independent pretty printer that can be used over and over again then to code a new pretty printer for each specific language that is to be pretty printed. [Ref. 10:p. 120]

For either type of syntax-directed pretty printer there are common issues that must be addressed. Issues such as where to add spaces and linefeeds, where to break a line, how to handle comments and what to do about syntax errors must be handled with great care [Ref. 10:p. 121]. The pretty printer developed in this thesis is a language dependent syntax-directed pretty printer (referred to throughout this thesis as a language dependent pretty printer). The generalization of this implementation is discussed and general rules for a language independent syntax-directed pretty printer are developed but are not implemented.

## D. SPEC

SPEC is a formal language for writing black-box specifications for components of software systems. SPEC uses the event model to define the black-box behavior of proposed and external systems. Black-box specifications are developed for the external interfaces of the system in the functional specification stage of software development, and for the internal interfaces in the architectural design stage. Discussion of the event model and the SPEC language, extracted from [Ref.1:pp. 3.1-3.15], follows. Appendix A [Ref. 2] contains a listing of the grammar for the SPEC language.

### 1. The Event Model

In the event model, computations are described in terms of events, modules and messages. An event occurs when a message is received by a module at a particular instant of time. A module is a black box that interacts with other modules only by sending and receiving messages. A message is a data packet that is sent from one module to another module.

Modules can be used to model external systems such as users and peripheral hardware devices, as well as software components. A module has no visible internal structure. The behavior of a module is specified by describing its interface. The interface of a module consists of the kinds of events that can occur at the module along with its response to each kind of event. Each kind of event corresponds to a different of incoming message. Each response consists of the later events directly triggered by a given initial event.

Any module accepts messages one at a time, in a well-defined order that can be observed as a computation proceeds. Message transmission is assumed to be reliable. Messages can have arbitrarily long and unpredictable transmission delays. The order of messages arriving is normally not under control of the designer.

In the event model each module has its own local clock. The local clocks of different modules are not necessarily synchronized with each other. Each event occurs at a well-defined instant of time, which is the time at which the destination module receives a message, according to its own local clock. The length of time between two events is precisely defined if both events occur at the same place. The length of time between two events at different locations can be estimated in terms of two readings of the same clock, but this is only an approximation because of unpredictable message delays in obtaining remote clock readings. The only kind of time interval meaningful in the event model is the duration between two events. There is no way to distinguish between computation delay and communication delay in the event model.

Each message has a sequence of zero or more data values associated with it. The other attributes of a message are its name, its condition and its origin. All of these attributes are single valued. Exceptions are modeled as messages by means of a condition attribute, which can take on the values "normal" and "exception". The condition of a message expressing a normal request for service is "normal". The condition of a message reporting an abnormal event somewhere is "exception", in which case the name of the message is the name of an exception condition.

The response of a module to a message is completely determined by the sequence of messages received by the module since it was created. A module is mutable if the response of the module to at least one message it accepts can depend on messages that arrived before the most recent incoming message. A module is immutable if the response of the module to every possible message is completely determined by the most recent incoming message. Mutable modules behave as if they had internal states or memory, while immutable modules behave like mathematical functions. A module is immutable if and only if it is not mutable.

Each module has the potential of acting independently, so that there is natural concurrency in a system consisting of many modules. Since events happen instantaneously and the response of a module is not sensitive to anything but the sequence of events at the module, the event module implies concurrent interactions with a module cannot interfere with each other at the level of individual events. This non-interference must be guaranteed by implementations which require a finite time interval to trigger the responses to an event. The response of a module is under the control of the designer.

In modeling concurrent systems it is sometimes necessary to specify atomic transactions. Atomic transactions are non-interruptible sequences of events at a module. Once a module starts an atomic transaction, it cannot accept any messages that are not part of the transaction until it is complete. Atomic transactions are sometimes needed to specify non-interference between concurrent sets of activities involving chains of multiple events at the same module. Atomic transactions must be used with care because they can lead to deadlocks if the protocols of the modules involved in a transaction are not compatible with each other, and can lead to starvation if a transaction goes on forever.

Modules can be used to model current and distributed systems, as well as systems consisting of a single sequential process. The event model helps to expose the parallelism inherent in a problem, because the only time orderings specified are those which are unavoidable and are agreed on by all observers.

Events can be triggered at absolute times. Such events are called temporal events. Temporal events are the means by which modules can initiate actions that are not direct responses to external stimuli. Formally a temporal event occurs when a module sends a message to itself at a time determined by its local clock. Unless explicitly stated otherwise, there may be an unpredictable delay between the time when the message is sent and the time when it is received, just like for any other message.

### 2. The SPEC Language

The SPEC language uses first order logic for the precise definition of the desired behavior of modules. The Spec language provides a means for specifying the behavior of three different types of modules:

(1) Functions
(2) State machines
(3) Types

Each of these types of modules is described in the following pages along with examples of each type of module.

a. Functions

Function modules are immutable and calculate functions on data types, where "function" is interpreted as in standard mathematics. Usually function modules provide only a single service and hence accept anonymous messages. Figure 2.12 gives an example of the specification for a square root function.

```
FUNCTION square_root{precision:real}
 WHERE precision > 0.0

 MESSAGE (x:real)
  WHEN x>= 0.0
   REPLY (y:real)
   WHERE y >= 0.0 & approximates (y*y,x)
  OTHERWISE REPLY EXCEPTION imaginary_square_root

 CONCEPT approximates (r1 r2:real)
  --True if r1 is a sufficiently accurate
  --approximating of r2.
  --The precision is relative rather than absolute
  VALUE (b:boolean)
   WHERE b<=> abs ((r1 - r2)/r2) <= precision
END

 Note: "--" introduces a comment and all keywords in
       Spec appear in all capital letters
```

Figure 2.12 Function Example

b. State Machines

A machine is a module with an internal state, i.e., machines are mutable modules. Figure 2.13 shows an example of a machine. The behavior of the machines is described in terms of a conceptual model of its state, rather than directly in terms of the messages that arrived in the past, because descriptions in terms of such a conceptual model are usually shorter and easier to read.

```
MACHINE inventory
  --assumes that shipping and supplier are other modules
 STATE (stock:map{item,integer})
 INVARIANT ALL (i:item::stock[1] >= 0)
 INITIALLY ALL (i:item::stock[1] = 0)

 MESSAGE receive (i:item,q:integer)
   --Process a shipment from a supplier.
   WHEN q > 0
    TRANSITION stock[1]=*stack[i] + q
     --Delayed responses to backorders are not shown.
   OTHERWISE REPLY EXCEPTION empty_shipment

 MESSAGE order (io:item,qo:integer)
   --Process an order from a customer.
  WHEN 0 < qo <= stock[io]
   SEND ship (is:item, qs:integer) TO shipping
    WHERE is = io, qs = qo
   TRANSITION stock[io] + qo = *stock[io]
  WHEN 0 < qo > stock[io]
   SEND ship (is:item, qs:integer) TO shipping
    WHERE is = is, qs = stock[io]
   SEND back_order (ib:item, qb:integer) TO supplier
    WHERE ib = io, qb + qs = qo
   TRANSITION stock[io] = 0
  OTHERWISE REPLY EXCEPTION empty_order
END
```

Figure 2.13 Machine Example

c. Types

A type module defines an abstract data type. An abstract data type provides many services therefore the messages of a type module usually have a name. An abstract data type consists of a set of instances and a set of primitive operations involving the instances. The instances are the individual data objects belonging to the type. The instances of an abstract data type are black boxes. The properties of the instances are not visible directly, and can only be observed and influenced by means of the primitive operations. The properties of an instance are determined by the primitive operation that created the instance and the sequence of primitive operations applied after it was created.

Date types are either mutable or immutable. For immutable types the set of instances and the properties of each instance are fixed. Operations producing instances of the type are viewed as selecting members of this fixed set. Figure 2.14 is an example of an immutable abstract data type.

The state of a mutable data type consists of a set of instances which have internal states. The initial state of a mutable type is an empty set of instances. Mutable types have operations for creating new instances, and usually also operations that can change the properties of an instance once it has been created. An example of a mutable abstract data type with immutable instances is the set of unique identifiers for the objects in a database.

```
TYPE rational
 INHERIT equality {rational}
 MODEL (num den:integer)
 INVARIANT ALL (r:rational::r.den ~= 0)

 MESSAGE ratio (num den:integer)
  WHEN den ~= 0
   REPLY (r:rational)
   WHERE r.num = num, r.den = den
  OTHERWISE REPLY EXCEPTION zero_denominator

 MESSAGE add (x,y:rational) OPERATOR +
  REPLY (r:rational)
  WHERE r.num = x.num*y.den+y.num*x.den,
        r.den = x.den*y.den

 MESSAGE multiply (x y:rational) OPERATOR *
  REPLY (r:rational)
  WHERE r.num = x.num*y.num, r.den = x.den*y.den

 MESSAGE equal (x y:rational) OPERATOR =
  REPLY (b:boolean)
  WHERE b <=> (x.num*y.den = y.num*x.den)
END
```

Figure 2.14 Immutable Abstract Data Type

An instance of a mutable data type is very similar to a state machine, except that the state machine is implicitly created at the start of the computation, while the instances of a mutable data type are created as a computation proceeds. A state machine has exactly one instance, while a mutable data type can have any number of instances. Figure 2.15 is an example of a specification of a mutable data type.

```
TYPE queue {t:type}
 INHERIT mutable {queue}
   --Inherit definitions of the concepts new and defined.
 MODEL (e:sequence)
   -The front of the queue is at the right end.
 INVARIANT tue
   --Any sequence is a valid model for a queue.

 MESSAGE create
   --A newly created empty queue.
  REPLY (q:queue{t}) WHERE q.e = []
  TRANSITION new(q)

 MESSAGE enqueue (x:t, q:queue{t})
   --Add x to the back of the queue.
  TRANSITION q.e = append([x], *q.e)

 MESSAGE dequeue (q:queue{t})
   --Remove and return the front element of the queue.
  WHEN not_empty (q)
   REPLY (x:t)
   TRANSITION *q.e = append (q.e,[x])
  OTHERWISE REPLY EXCEPTION queue_underflow

 MESSAGE not_empty (q:queue{t})
   --True if q is not empty.
  REPLY (b:boolean) WHERE b <=> (q.e ~= [])
END
```

Figure 2.15 Mutable Abstract Data Type

# III. DESIGN AND IMPLEMENTATION

The actual design and implementation for the pretty printer was motivated not only on the specific application to this particular language but also by the desire to generalize the solution to apply to other languages. It is highly desirable that what is learned from this particular case can be extended to the design of a language independent pretty printer.

## A. DESIGN QUESTIONS

A language dependent pretty printer is a software tool to increase the readability and understandability of a specific formal language. In this light the design questions must be centered around increasing both the readability and understandability of Spec (the language used in this application).

One important constraint related to this specific application must be considered carefully. The Kodiyak compiler cannot be changed. There are provisions to add features to the compiler but the overall design and implementation of the Kodiyak compiler is fixed. Therefore any design decisions must not require any modifications to the Kodiyak compiler itself.

### 1. Specific Design Issues

There are five considerations specific to this implementation that must be addressed. They are:

(1) Length of each line
(2) Standards for indentation
(3) Token length
(4) Comments
(5) Keywords

#### a. Line Length

The length of the line defines the maximum number of characters that can be printed on any given line. This length can be chosen regardless of the width of the output medium but must permit the maximum number of characters to all be printed within the output medium's width. Depending on the implementation this can be either a global constant or an input parameter.

#### b. Indentation

Indentation, as applied to computer programs, groups together lines of related code. An example in the English language of indenting is a formal outline. There are major topic headers. Under each major topic header is subtopic headers with each subtopic being subdivided (depending on how detailed the outline is). Figure 3.1 is an example of a simple formal outline.

As Figure 3.1 shows indentation makes the structure easy to see. Related items all start at the same distance from the left margin. As the number of subdivisions grow the longer the indentation (relative to the left margin).

```
  I.  Introduction
      a.  Software engineering
          1.  General introduction
      b.  Functional specification
          1.  Nonformal
          2.  Formal
              (a)  Spec Language
 II.  Background
      a.  Attribute grammar
          1.  Definition
          2.  Purpose/role
          3.  Attributes in general defined
              (a)  Synthesized
              (b)  Inherited
      b.  Kodiyak
          1.  Definition
          2.  Format
          3.  Semantics
```

Figure 3.1 Sample Outline

In computer programs how much to indent related lines is an important question. Using too many spaces for indenting can easily run lines of code off a page. On the other hand not using a wide enough indent does not show the structure of the code and decreases readability. A compromise must be made between readability and losing lines of code off a page. The standard for program code is between two and five spaces for indenting each subdivision of related lines of code.

c. Token Length

Token length is the actual length of any token used in the language. One assumption that should be made is that no token is longer than the maximum number of characters allowed on one line.

d. Comments

Comments are added to provide documentation to the code and also explain what the code is actually doing. Therefore it is extremely helpful to have comments disbursed among the lines of code. For readability comments should not appear between code segments that are on the same line. The particular language implemented will specify the allowable placement of comments and how the comments are identified.

e. Keywords

The final consideration is keywords. Does the language contain keywords? If so, how are they to be distinguished? Are they unique (i.e., reserved) or can the programmer use a language keyword with a totally different meaning? Do keywords have special format? Some of the possibilities are:

(1) all capitalized with the rest of the code lowercase
(2) all capitalized with the rest of the code a combination of lowercase and uppercase
(3) underlined
(4) proceeded by a special character

### 2. General Design Issues

When considering the specific questions generic rules should also be a consideration. Can this specific implementation be easily modified to handle different/slight modifications? For example if the line length is increased will this radically effect the software or is it a very easy change? Additionally, can more features be added if desired? Can debugging aids be added to the already existing software? Will any modifications still be compatible with the existing software? Will added features effect the original code causing a revision? If these added features cause the original code to be revised, how much work is involved and are the added features worth the added work?

## B. DESIGN DECISIONS

The decisions that were made for this specific implementation concern the questions raised in the previous pages. They are:

(1) Length of a line
(2) Token length
(3) Indentation
(4) Comments
(5) Keywords

### 1. Length of Line Decision

The length of the standard line for this implementation is 80 characters. It makes no difference whether the output from the pretty printer is printed on 8

1/2" paper or the wider 14" paper. This print out will fit on either size of paper which allows for more flexibility in what printer and paper is used. The only drawback to this decision is that if the wider paper is used the right 5 1/2" of the paper will not be utilized.

### 2. Indentation Decision

The standard indentation is always three blank characters. As each sentence in the language is subdivided (broken down into the grammar rules) the indentation is expanded by an additional three blank characters. Three is a fairly reasonable number between the standard, in computer science, two and five spaces. It allows the reader's eyes to see what sentences and parts of sentences are all related at the same level. Additionally, there is not too much indenting (i.e., using five blank characters) so that if a lot of subdividing (or recursion) occurs the indenting will not run the print out off the page.

### 3. Token Length Decision

One necessary assumption made by this implementation is that the token length for any token will not exceed the maximum line length. Any token greater than 80 can never be printed with the restriction placed on this implementation. If the line length is increased then the maximum token length can also be increased.

4. Comment Decision

Comments are allowed to be one line or multiple lines long. Comments can come at the end of a line of code filling the space until the right-hand margin is reached, or can be one line long starting at the left-hand margin, or can extend over several lines with each new line flush with the left-hand margin.

The one restriction on a comment is that it is always started with a special character and its total length (including the special character) is less than or equal to the line length. For this implementation the special character to introduce a comment is "--".

For a comment extending beyond the line length it must be broken up into two lines each starting with a special character. If this is not done by the user part of the comment may be lost when it is printed. It is important to emphasize that it is the user's responsibility to insure that comments are not longer then 80 characters since this pretty printer implementation assumes a single comment will fit on one line (80 characters or less).

For output the special comment character is always followed by one blank character. If a comment comes at the end of a line of code the pretty printer will place two blank characters before the special character. If the comment starts a new line there will be no proceeding blank characters. Figure 3.2 shows examples of comments.

```
1.
     -- This comment is a single line comment.
2.
     expression = exp + exp  -- Comment following code
3.
     expression = exp + exp  -- Comment following code
     -- but this time code extends more than one line.
4.
     -- This is a sample of a multiline comment with
     -- the first line being flush with the left margin.
```

Figure 3.2 Comment Examples

### 5. Keyword Decision

All keywords in the output are capitalized. In this implementation there are three types of keywords. The importance of these three types will be explained in more detail later. Figures 3.3, 3.4 and 3.5 show the three types of keywords. All three categories of keywords are typed in all uppercase letters with the difference coming in the indentation rules related to the keywords.

## C. ATTRIBUTE DEFINITIONS

The design of the pretty printer centers around the selection of attributes. The goal of the design is to create a software package that produces a formatted output that is readable and reflects the structure of the original code. The format of the input must be irrelevant to the pretty printer. It must also be noted that the input must be syntactically correct for the pretty printer to operate correctly.

```
FUNCTION       ITERATOR
END            TEMPORAL
MACHINE        EXCEPTION
TYPE           THEN
DEFINITION     ELSE
INSTANCE       ELSE_IF
INHERIT        VALUE
HIDE           DO
RENAME         IF
IMPORT         TRANSACTION
FROM           INITIALLY
EXPORT         INVARIANT
MESSAGE        STATE
WHEN           MODEL
OTHERWISE      CONCEPT
CHOOSE         FOREACH
REPLY          SUCH
GENERATE       TRANSITION
SEND           VIRTUAL
TO             WHERE
```

Figure 3.3 General Keywords

```
TIME           ALL
DELAY          SOME
PERIOD         NUMBER
ALL            SUM
PRODUCT        SET
MAXIMUM        MINIMUM
UNION          INTERSECTION
NANOSEC        MICROSEC
MILLISEC       SECONDS
MINUTES        HOURS
DAYS           WEEKS
OPERATOR
```

Figure 3.4 Expression Keywords

```
AS
OD
FI
```

Figure 3.5 Special Keywords

The attributes will define the language dependent pretty printer. There should be as few attributes as possible with each limited to one specific function therefore making each very limited in scope. All the attributes can be one of two types: synthesized attributes or inherited attributes. Synthesized attributes are based on the attributes of the descendants of the nonterminal symbol. Inherited attributes are based on the attributes of the ancestors. Synthesized attributes are evaluated from the bottom up in the tree structure, while inherited attributes are evaluated from the top down. [Ref. 8:p. 130]

The pretty printer will need values for the print head position, lengths of symbols, values for indenting and the actual string that will be printed. With these ideas in mind this implementation utilizes six attributes as shown in Figure 3.6.

The type of attribute, the type of value the attribute is and a definition (with examples if necessary) of each attribute follows. Remember that each attribute has one unique purpose which is very limited in scope.

| ATTRIBUTE | DEFINITION |
|---|---|
| bcursor | beginning cursor position |
| ecursor | end cursor position |
| padding | blank spaces to pad beginning of line |
| indent | indentation |
| str_value | string value |
| length | number of characters long |

Figure 3.6 Pretty Printer Attributes

1. Bcursor

Bcursor is short for beginning cursor position. It is the column position at which the left-most character of a production rule will be printed. This is an inherited attribute with an integer value. As the print head moves across the paper from left to right the value of bcursor will increase from one to the maximum line length (in this implementation maximum line length is 80). Figure 3.7 shows examples of bcursor.

2. Ecursor

Ecursor is short for end cursor position. It is the column position at which the right most character of a production will be printed. This is a synthesized attribute with an integer value. This attribute can range in value from one to the maximum line length (in this implementation line length is 80). The ecursor of any rule is the bcursor of the next rule in the parse tree. Figure 3.7 shows the interplay between bcursor and ecursor.

```
action_list
   :  EXCEPTION parametrized_name
   {
    parametrized_name.bcursor=action_list.bcursor+10(*);
    action_list.ecursor = parametrized_name.ecursor;
   }

message_header
   :  optional_exception optional_name formal_arguments
   {
    optional_exception.bcursor = message_header.bcursor;
    optional_name.bcursor = optional_exception.ecursor;
    formal_arguments.bcursor = optional_name.ecursor;
    message_header.ecursor = formal_arguments.ecursor;
   }

  (*)10 is the number of characters in the word
   EXCEPTION plus one
```

Figure 3.7 Bcursor and Ecursor Examples

## 3. Padding

Padding is short for necessary blank spaces to pad. It is a string of blank spaces to put at the beginning of a new line when the current line must be split because it is longer than the maximum line length. This is an inherited attribute with a string value of blank characters. The value of the attribute can range from zero blank characters to the maximum line length. Figure 3.8 shows an example of the implementation of the padding attribute.

```
instance
   :optionally_virtual INSTANCE parametrized_name =
    parametrized_name comment hide renames END
   {
    parametrized_name[1].padding =
      [spaces(optionally_virtual.ecursor),spaces(9)];(*)
    parametrized_name[2].padding =
      parametrized_name[1].padding;
    hide.padding = hide.indent;
    renames.padding = renames.indent;
   }

(*)spaces(9) is a function that produces a string of nine
   blank characters
```

Figure 3.8    Padding Example

### 4. Indent

Indent is short for indentation. It is a string of blank characters associated with each nonterminal symbol of the same production rule. At each level of nesting the number of blanks associated with the value of the attribute indent increases by three. This is an inherited attribute with a string value. The string value for indent is the number of blanks associated with a nonterminal symbol. The length of the string can range from zero to the maximum length of line (in this case it also has to be a multiple of three so the maximum value would be 78). This allows all related lines to be easily seen and to maintain the structure of the code. Figure 3.9 shows an example of how the indent attribute is used.

```
function
   : optionally_virt interface messages concepts
   {
    optionally_virt.indent=[function.indent,spaces(3)](*);
    interface.indent = [function.indent,spaces(3)](#);
    messages.indent = [function.indent,spaces(3)];
    concepts.indent = [function.indent,spaces(3)];
   }
```

(*)"[" and "]" are symbols to represent string concatenation of all values between the two symbols

(#)spaces(3) is a function producing a string of three blank characters

Figure 3.9 Indent Example

On the surface it appears that indent and padding are very similar and could be combined. This is not the case. Indent has a length that is always a multiple of three, a forced linefeed occurs with every indent and indent is only used with keywords. Indent is designed to show the nesting levels of all symbols from the same production. Padding, on the other hand, can range in length from zero to the maximum line length, linefeeds are optional and is used only to assist in lines that are too long to fit on one line. Padding is designed to format long lines keeping all text of the long line grouped together. Figure 3.8 shows the interplay between the two attributes padding and indent.

5. Str value

Str_value is short for string value. The output of the pretty printer is the str_value attribute of the start symbol at the root of the parse tree. It is the set of terminal symbols derived from a production rule together with spaces and linefeeds for formatted output. This is a synthesized attribute with a string value. The length of str_value can be of any value from zero to infinity. The str_value of the start symbol will have the longest length. Concatenation is used to put different str_value attributes together. Figure 3.10 shows an example of the concatenation of strings to obtain a value for str_value.

```
definition
   : DEFINITION interface concepts END comment
   {
    definition.str_value = ["\n", definition.indent,
       "DEFINITION ", interface.str_value,
       concepts.str_value,"\n", "END", comment.str_value,
       "\n","\n"];
   }
```

Figure 3.10   Str_value Example

6. Length

Length is short for number of characters long. It is the number of printable characters in a given production rule. It is a synthesized attribute with an integer value, which is used to determine if an expression will fit on the remainder of the current line. Length is important because it counts the actual number of characters ignoring possible padding, carriage returns, or line feeds. It is utilized by the expression production rule and production rules that have comment as one of their nonterminal symbols. Figure 3.11 shows examples of the attribute length.

```
expression
   : '$' expression
   {
      expression[1].length = 1 + expression[2].length;
      expression[2].bcursor = expression[1].bcursor + 1 +
                              expression[2].length <= 80
         ->  expression[1].bcursor + 1
         #   len(expression[1].padding) + 1;
   }
```

Figure 3.11 Length Example

D. PRETTY PRINTER RULES

The language dependent pretty printer has only a few general rules of interest to the user of the pretty printer. For an implementor or someone who wants to know more details about the pretty printer, the specific rules define the behavior of the pretty printer in detail.

1. Rules for Using the Pretty Printer

There are several general rules for the use of this pretty printer. First the input must be syntactically correct. If it is not correct the software will print out a syntax error message (it may or may not print out any of the input data). Figure 3.12 shows an example of what happens when the pretty printer is supplied with syntactically incorrect code.

```
        1: MACHINE
                ^Syntax error

    Note:  1 is the line number of the error
           ^ points to the point the syntax error
             was detected
```

Figure 3.12   Syntactically Incorrect Code Output Example

Secondly, the input can come from either a file or can be typed from the terminal. The input code can be in any format provided its syntax is correct. For most applications it seems quite reasonable for the user to already have a file created. It is extremely time consuming to manually enter the code each time, not to speak of the likelihood of typing mistakes which will force the user to start over again. It is highly recommended that the input come from a file.

The pretty printer is invoked in one of two ways. The first method is by typing the file name of the compiled Spec code followed by a file name with a set of options. The possible options that are available are shown in Figure 3.13 [Ref. 9:pp. 23-24].

The second method to invoke the pretty printer is to type the file name of the compiled Spec code, a carriage return and then manually enter all of the code for the input from the keyboard. When finished type "control d" and the output will appear at the standard output. Figure 3.14 shows an example of executing the pretty printer with both of the methods described. The name of the compiled Kodiyak code for the pretty printer is stored in the file printer and the name of the input file is SAMPLE (when one is specified). An explanation of the different commands invoked follow Figure 3.14.

| | |
|---|---|
| -h | Print out a list of legal options. |
| file | Read input from "file"rather than the standard input |
| -e | Continue attribute evaluation even after an error occurs. This is useful when debugging attribute definitions. |
| -l | Print out all tokens as they are scanned. |
| -y | Print out all grammar rule reductions as they occur. |
| -L | Turn on LEX's debugging features. |
| -Y | Turn on YACC's debugging features. |
| -c | Generate a core image when a run-time error occurs |
| -s | Print out storage statistics after all attribute evaluations is completed. |
| -o file | Divert the standard output to "file". |

Figure 3.13 Printer Options

```
A
   printer SAMPLE
B
   printer SAMPLE -o OUTPUT
C
   printer -h
D
   printer  SAMPLE -y
E
   printer
```

Figure 3.14 Invoking the Code Example

Part A invokes the pretty printer using the input file SAMPLE and the formatted output will be printed to the standard output. Part B invokes the pretty printer using the input file SAMPLE and the formatted output will be sent to the file OUTPUT. Part C invokes the pretty printer but in this case a list of the options that are available will be printed. The list printed is similar to Figure 3.13. Part D invokes the pretty printer using the input file SAMPLE but in this case it will print out all grammar rule reductions as they occur. This may assist in diagnosing syntax errors. Part E invokes the pretty printer with the input coming from the terminal. The user must type in the necessary code followed by a "control d" to exit. Output will go to the standard output.

## 2. Rules for Implementing the Pretty Printer

The above section looked at the general rules for the successful operation of the pretty printer. Those are the only rules the user needs to know to use the pretty printer. An implementor or someone who may want to know more details will be interested in the specific rules for the pretty printer. There are four specific implementation rules. These rules with their guidelines and exceptions explain the complete operation of this language dependent pretty printer. The four rules that will be explained in detail deal with:

(1) Keywords
(2) Terminal symbols
(3) Nonterminal symbols
(4) Comments

### a. Keyword Rule

Keywords are special reserved words in Spec. All keywords are capitalized. All the other tokens in the language can use uppercase or lowercase letters or a combination of the two, but all keywords are distinguished by the use of all uppercase letters. Also as noted earlier there are three types of keywords:

(1) General keywords
(2) Expression keywords
(3) Special keywords

The rule for general keywords states the output consists of a carriage return, line feed, current production rule indenting and then the keyword. The rule for expression keywords states that these symbols are to be treated exactly the same as a terminal symbol. See the following section for the terminal symbol rule.

All special keywords appear after a general keyword. RENAME always proceeds AS, IF always proceeds FI and DO always proceeds OD. For these special keywords the rule states if room allows the special keyword will appear on the same line as the general keyword otherwise the special keyword will appear directly beneath the general keyword (on the following line) at the same degree of indentation.

The exceptions to the above three rules are few but are extremely important to the pretty printer. First, only the first general keyword is effected by the rule for general keywords when two general keywords appear one after the other. This saves an unnecessary carriage return and line feed. Second, all three rules are ignored when any type of a keyword directly follows a terminal symbol. In this case the rule for that keyword type will be followed only if the keyword will not fit on the current line. Figure 3.15 summarizes the three keyword rules and the exceptions to these keywords rules.

#### b. Terminal Rule

A terminal symbol is defined as a symbol that can appear only on the right side of a production rule [Ref. 6:p. 97] or as a primitive symbol of the language [Ref. 5:p.

77] (i.e., cannot be reduced any further). Keywords by this definition are terminals but in this implementation keywords are considered a separate category of symbols.

RULE 1
The rule for a general keyword states the output consists of a carriage return, a line feed, an associated production rule indentation and the keyword (with the keyword all in capital letters).

RULE 2
Expression keywords are treated like terminal symbols. (See terminal symbol rule).

RULE 3
All special keywords follow a general keyword. The special keyword appears on the same line as the general keyword if room permits; otherwise, the special keyword will appear directly beneath the general keyword at the same degree of indentation

EXCEPTIONS

1.
When two general keywords appear in a row only the first general keyword follows the rule for general keywords. This rule is ignored by the second keyword.

2.
When any keyword appears after a terminal symbol all three rules are ignored. The rules are only followed if the keyword will not fit on the current line.

Figure 3.15 Keyword Rules and Exceptions

In this implementation there are two types of terminal symbols. They are constant length terminal symbol (CL terminal) and variable length terminal symbol (VL terminal). The CL terminal symbols (those with symbol values different from their symbol names) are shown in

Figure 3.16 and the CL terminal symbols (those with the same symbol name and value) are shown in Figure 3.17. The VL terminal symbols involve the use of the built in attribute %text. The six VL terminal symbols along with the way they appear in the language are shown in Figure 3.18.

| Name | Value | Name | Value |
|---|---|---|---|
| AND | & | IFF | <=> |
| OR | \| | NOT | ~ |
| IMPLIES | => | LE | <= |
| GE | >= | NE | ~= |
| NLT | ~< | NGT | ~> |
| NLE | ~<= | NGE | ~>= |
| EQV | == | NEQV | ~== |
| RANGE | .. | APPEND | \|\| |
| ARROW | -> | MOD | \\ |
| EXP | ** | BIND | :: |

Figure 3.16 CL Terminal Symbols
Symbol Values Different from Symbol Name

The terminal symbol rule checks for the end of line. When a terminal symbol (either kind) is encountered an end of line check is done. This length check is to see if the maximum line length will be exceeded if the terminal symbol is added to the str_value. This general rule for terminal symbols can be divided into two cases depending on the value of the length check sum. If this length check sum is less than or equal to the line length than the value of the production rule attribute str_value is the value of the terminal symbol (depending on the terminal symbol it may be concatenated with one blank character). If the length check sum is greater than 80 then the value of the production rule attribute str_value is the concatenation of a carriage return, line feed, production rule padding and the value of the terminal symbol (depending of the terminal symbol it may be concatenated with one blank character).

| | |
|---|---|
| = | < |
| ( | > |
| ) | - |
| : | + |
| ; | * |
| { | / |
| } | U |
| , | . |
| ? | ] |
| $ | [ |
| @ | ! |
| # | MOD |
| IN | |

Figure 3.17 CL Terminal Symbols
Symbol Name Same as Symbol Value

| Terminal Symbol Name | Appearance in Code |
|---|---|
| NAME | NAME.%text |
| INTEGER-LITERAL | INTEGER-LITERAL.%text |
| REAL-LITERAL | REAL-LITERAL.%text |
| CHAR-LITERAL | CHAR-LITERAL.%text |
| STRING_LITERAL | STRING-LITERAL.%text |
| COMMENT | COMMENT.%text |

Figure 3.18 VL Terminal Symbols

The length check sum is computed in one of two ways depending upon which production rule is being used. If the terminal symbol (either kind) is not part of an expression production than the length check sum is the sum of the current cursor position, the length of the terminal symbol and one (for a blank space). Otherwise, if the terminal symbol (either kind) is part of the expression production rule then the length check sum includes the sum of the current cursor position, the length of the terminal symbol value, one (for a blank character) and the length of the symbol or symbol to the right of the terminal symbol in the production rule. Figure 3.19 summarizes the general rule for the terminal symbols along with the rules for calculating the length check sum.

There are three exceptions to the terminal symbol rule. First when a nonterminal symbol (a symbol that can be reduced) precedes a VL terminal symbol (which precedes a CL terminal symbol), the length check sum includes the length of the VL terminal symbol as well as the CL terminal symbol.

The second exception to this rule is when CL terminal symbols appear in pairs. The only CL terminal symbols this exception applies to are the right and left parenthesis and the right and left square bracket. The right parenthesis and the right square bracket do not cause a check for the end of line.

```
The general rule when encountering a terminal symbol
(both CL & VL) is to do a check for the end of the line.


LENGTH CHECK SUM <= 80
   str_value = terminal symbol value (may have one blank
               character at end)

LENGTH CHECK SUM > 80
   str_value = [carriage return, line feed, production
                rule padding, terminal symbol value]
               (may have one blank character at end)


TERMINAL SYMBOL PART OF EXPRESSION PRODUCTION RULE
   length check sum = current cursor position +
                      length of terminal symbol +
                      (possibly one for blank space)


TERMINAL SYMBOL NOT PART OF EXPRESSION PRODUCTION RULE
   length check sum = current cursor position +
                      length of terminal symbol +
                      length of rule(s) to the rule of
                      the terminal symbol +
                      (possible one for blank space)
```

Figure 3.19 Terminal Symbol Rule and Length Check Sum Calculation

The third exception occurs when a comment immediately precedes a terminal symbol (either kind). A check is first done to see if a comment exists. If a comment does not exist the general rule is followed, otherwise the production rule str_value includes a carriage return, line feed, production rule padding and the value of the terminal symbol (depending on the terminal symbol may include one blank character). Figure 3.20 outlines the three exceptions to the general rule. Figures 3.21 and 3.22 show examples of the terminal symbol rule and exceptions.

```
1.  Nonterminal symbol proceeding a VL terminal
    symbol (with the VL terminal symbol proceeding
    a CL terminal symbol)

  length check sum = current cursor position +
                     length of the VL terminal symbol +
                     length of the CL terminal symbol +
                     one (blank between terminal
                     symbols)


2.  CL terminal symbols appearing in pairs
    (applies to ")" and "]" only)

  These two symbols do not cause a end of line check


3.  Comment immediately precedes a terminal symbol

   If comment does not exist follow general rule
   else
     str_value = [carriage return, line feed,
                  production rule padding,
                  terminal symbol value]
                  (possibly one blank character
```

Figure 3.20 Terminal Rule Exceptions

c. Nonterminal Rule

The rule for nonterminal symbols is the easiest of all the rules. A nonterminal symbol in a production rule will specify the value of all attributes that its own production rule needs and those values of the attributes that its children's production rules may need. Depending on which nonterminal symbol is involved any or all of the six

```
A.  Length check with expression production

  expression
     : NOT expression
      {
       expression[1].str_value = expression[1].bcursor +
         1 + expression[2].length  <= 80
       ->  ["~", expression[2].str_value]
       #   ["\n",expression[1].padding,"~",
              expression[2].str_value];
      }

B.  Length check without expression production

  field_list
     : field_list ',' field
      {
       field_list[1].str_value =
         field_list[2].ecursor + 2 <= 80
       ->  [field_list[2].str_value,", ",field.str_value]
       #   [field_list[2].str_value,", ", "\n",
             field_list[1].padding,field.str_value];
       }

C.  Length check with first exception

  expression
      : expression '.' NAME
       {
        expression[1].str_value = expression[2].ecursor +
          1 + len(NAME%text) <= 80
        ->  [expression[2].str_value, ".",NAME.%text]
        #   [expression[2].str_value, ".","\n",
              expression[1].padding,NAME.%text];
       }
```

Figure 3.21   Terminal Symbol Examples

A. Length check with second exception

```
actual_parameters
     : '(' arg_list ')'
      {
       actual_parameters.str_value =
         actual_parameters.bcursor + 1 < 80
       ->  ["(", arg_list.str_value,")"]
       #   ["\n",actual_parameters.padding, "{",
            arg_list.str_value,"}"];
       }
```

B. Comment before terminal symbol

```
concept
   :  CONCEPT formal_pa ':' type_spec
    {
     concept.str_value = formal_pa.ecursor < 0*
     ->  ["\n","\n",concept.indent,CONCEPT,
          formal_pa.str_value,"\n",forma_pa.padding,
          ": ",type_spec.str_value]
     #  formal_pa.ecursor + 2 <= 80
     ->  ["\n","\n",concept.indent,CONCEPT,
          formal_pa.str_value,": ",
          type_spec.str_value]
     #   ["\n","\n",concept.indent,CONCEPT,
          formal_pa.str_value,"\n",forma_pa.padding,
          ": ",type_spec.str_value];
     }
```

Figure 3.22   Terminal Symbol Examples

attributes that this implementation uses can be specified. Not all of the nonterminal symbols use all of the six attributes. Figure 3.23 shows some examples of how attributes are used with the nonterminal symbols. In the first example in Figure 3.23 the nonterminal symbol "name_list" only needs two attributes and "comment" needs only one attribute. In the second example in Figure 3.23 the nonterminal symbol "interface" does not need or use the attribute indent but the nonterminals "imports", "inherits" and "export" need indent therefore the parent is passing along its value of indent to its children.

#### d. Comment Rule

The rule for a comment concern where a comment can and cannot be placed. This rule is language dependent. In Spec comments can occur after any nonterminal symbol but comments cannot occur immediately after any terminal symbol. See Figures 3.16, 3.17 and 3.18 for the list of CL terminal symbols and values and VL terminal symbols. There are two exceptions to this rule. First, a comment can occur after the CL terminal symbol that comes as the second in a pair (i.e., ")" and "]" ) when the production rule that involves the CL terminal symbol does not have a nonterminal symbol immediately following the CL terminal symbol. Second, a comment cannot occur immediately after a keyword. Figure 3.24 summarizes the comment rule and its two exceptions. Note this is strictly dependent on the language implemented.

```
  A nonterminal symbol will specify the value of all
  attributes that its own production rule needs and
  those attributes that any of its children may need.

EXAMPLE ONE

hide
  :  HIDE name_list comment
   {
    name_list.bcursor = hide.bcursor + 5;
    name_list.padding = [hide.padding, spaces(5);
    comment.bcursor = name_list.ecursor;
    hide.str_value = ["\n",hide.indent,"HIDE ",
                      name_list.str_value,
                      comment.str_value];
    }

 EXAMPLE TWO

 interface
   :  NAME inherits imports export comment
    {
      inherits.indent = [interface.indent, spaces(3)];
      export.indent = [interface.indent, spaces(3)];
      imports.indent = [interface.indent, spaces(3)];
      interface.str_value = [NAME.%text,
                             inherits.str_value,
                             imports.str_value,
                             export.str_value,
                             comment.str_value];
    }
```

Figure 3.23   Nonterminal Rule and Examples

A comment can occur after any nonterminal symbol but cannot occur immediately after any terminal symbol (either kind).

EXCEPTIONS

A. A comment can occur after the CL terminal symbol ")" or "]" if the production rule does not have a nonterminal symbol immediately following either of these CL terminal symbols.

B. A comment cannot occur immediately after any keyword.

Figure 3.24 Comment Rule

# IV. CONCLUSIONS

The conclusions of this thesis address the issues of the specific implementation of one language dependent pretty printer and how this can be extended to generate a general purpose language independent pretty printer. What was learned through the development, design and implementation of this pretty printer can be abstracted, generalized and expanded to create a language independent pretty printer.

## A. IMPACT OF DESIGN DECISIONS

The major impact on any and all of the design decisions relates to readability, understandability, portability and ease in modifying. Can what was done for a single implementation be of any great value? The answer to this question is yes. Can all the rules and exceptions be abstracted and extrapolated? The issues surrounding the design decisions made in this particular implementation and the ability to extend what was learned center around four key ideas. These four key ideas are:

(1) Global parameters
(2) Attributes
(3) Standardization
(4) Comments

### 1. Global Parameters

In the normal sense of global parameters, such as in a Pascal program, this implementation does not have any global parameters. There is no method available to declare a global parameter at the beginning of the program due to the tree like structure that is used in parsing and evaluating the pretty printer. On the other hand this implementation does have "global parameters" to a limited degree. The line length and therefore the right-hand margin is set on 80. Any reference to the right-hand margin is made in reference to this global parameter. One global change to the value 80 (with an editor) can globally change the width of the print out and thereby changing the value of the right-hand margin.

The global indentation change can also be easily made. SPACES(3) is used throughout to handle the indentation. If the indentation width was changed to four, for instance, one simple global change with an editor could change all SPACES(3) to SPACES(4).

The use of global parameters as explained above increases the portability of the code. Additionally, it increases the adaptability and ease of modification of this language dependent pretty printer to different users preferences and desires. The pretty printer code can also be modified to accept variable parameter values from the command line if the computer system utilized supports that feature (i.e., printer -l 80, printer -s 4). In this case Kodiyak would have to be modified to support the option of command line inputs.

## 2. Attributes

There are only a small number of attributes. These six attributes are:

(1) Bcursor
(2) Ecursor
(3) Padding
(4) Indent
(5) Str_value
(6) Length

Each attribute is unique and handles one specific well defined function. If one attribute (or the concept behind it) is changed or altered the other attributes will not be affected. This is true for all except bcursor and ecursor. These two attributes go hand in hand and changing one will greatly affect the other.

## 3. Standards

There is standardization of the implementation throughout the entire code for the pretty printer. With a given number of general rules and a few exceptions the implementation follows a fairly standard organization (i.e., each production rule is basically implemented in the same way). With standardization of rules generic rules can easily be derived from the specific rules.

In addition to standardization in the pretty printer code, there are also standards in computer science that are adopted by the implementation. This includes the rule of thumb standard for the length of the standard indentation unit.

#### 4. Comments

Allowing for freedom of style in comments allows comments to be almost any place within the code. Comments add tremendously to the overall readability of program code. The implementation allows for the widest variety of comments within reason. Within reason means that there are times when you do not want a comment. As an example a comment is not a good idea in the middle of a mathematical expression.

## B. EVALUATION OF THE PRETTY PRINTER

The qualities of the pretty printer deal with the software, documentation and devices used in the actual development and execution of the pretty printer. The qualities to be concerned with center around the following three categories:

(1) Kodiyak compiler
(2) Syntax errors
(3) Software extensions

#### 1. Kodiyak Compiler

The Kodiyak compiler is a complex piece of software but when modifications were made there was no apparent change in the efficiency and no slow down in the processing time. To make a change to the compiler one of its many related files (as shown in Figure 4.1) was modified and then the compiler was recompiled. Recompiling is a small sacrifice when making changes to an extremely complicated piece of software.

| NAME | PURPOSE |
|---|---|
| locallib.c | helper functions for the Kodiyak compiler |
| man.entry | Unix Programmer's Manual |
| k | software driver |
| kclib.c | library functions and C definitions |
| kmain.c | main routine for Kodiyak programs |
| kodiyak.k | Kodiyak translator |
| kodiyak.m4 | Kodiyak translator |
| kscript | executes the translator |
| mac.m4 | macros |

Figure 4.1 Kodiyak Files

With everything that is good there are also some drawbacks. Changes to the compiler are easy to make, but sometimes it requires a change to the Kodiyak code with an associated recompiling of that code and other times the change requires a modification to the file that controls the overall execution of the Kodiyak compiler (file named k). Figure 4.2 shows some examples of these situations with the resulting actions needed to correct the problems.

Another area of concern with the Kodiyak compiler is the error messages generated. Understandably the Kodiyak code and documentation was written by one person within a six month time frame. Even so the error messages generated are hard to understand and even harder to correct. There is a lack of standardization when an error had to be corrected. The user has to search through the multiple related files (those listed in Figure 4.1) to find the solution to a given error. Once the correction to the error has been found the most obvious correction may not fix the problem.

1. Table overflow
   Change the size of the table in file k

2. Memory overflow
   Change the value of associated variable in file kclib.c

Figure 4.2 Compiler Change Examples

As an example consider the following. An error appeared that stated "OUT OF XNODESPACE". The solution was found in file kclib.c. The documentation [Ref. 11] stated:

> "The following definitions may be over-ridden from the C compiler's command line. This allows users to increase or decrease the space allocated to each data type according to his needs. cc -DXPAIRSPACE=50000 agprog.c -o agprog would give the user's program 50,000 pairs to use instead to the 20,000 allocated by default."

This feature did not work as advertised. After trial and error the solution found involved changing the value of XNODESPACE in two separate locations in the file kclib.c and then recompiling the Kodiyak compiler. That solution was not written anywhere or even suggested.

2. <u>Syntax Errors</u>

Syntax errors are concerned with the errors in the syntax of the input code. These syntax errors can basically be of two types. There are errors caused by typing mistakes and errors concerned with missing grammar rule(s) in the language. These syntax errors can be either easy or hard to find.

When the Kodiyak compiler finds any type of error it doesn't guess what the user meant. It crashes and prints an error message. The difficulty involves tracking down the actual error. Figure 4.3 shows examples of typical error messages.

1. Spelling or typing error

```
1:  MESAGE
          ^Syntax error
```

2. Grammar rule missing/Input does not match grammar

```
3:  MESSAGE FUNCTION
                  ^Syntax error
```

Figure 4.3 Typical Error Messages

As Figure 4.3 shows the error messages can be quite cryptic. In all cases the error occurs somewhere after the place that the syntax error pointer points. The error in fact may be a several lines further down in the code because the language is parsed in a tree like structure. The error message pointer points to the production rule that the pretty printer code was parsing at the time the error occurred (the actual error can be a descendant of the production rule).

3. Software Extensions

Software extensions are concerned with adding software to the actual Kodiyak software. The Kodiyak software is quite adaptable and has allowed for user defined functions and applications to be added to the existing software. The only drawback to this is that the user defined software must be added to the end of an already existing set of library functions. Through experience it has been determined that if the user-defined functions are written in a separate file and declared in an option added to the file k (that directs the Kodiyak) incompatibility error messages will appear. If the same code is placed at the end of the file kclib.c the Kodiyak has no problems with the user defined functions and everything runs smoothly.

One additional note about the user defined functions. This allows the user to write any type of function that is desired as long as it is written is C. With this implementation a new function (named SPACES) was created to change an integer into the corresponding number of blank characters. Writing a fairly efficient small loop function seemed quite easy until the code was compiled. Then the problem of incompatibility arose between the existing code and the new function written in C. To solve this compatibility problem an inefficient function was written to handle the conversion of integers zero through 80 to a corresponding string of blank characters. If the line length is increased above 80 the function SPACES will have to modified to handle all integer values greater than 80.

## C. ANALYSIS OF CODE

The analysis of the pretty printer code has to look at three general areas. These three areas are:

(1) Efficiency of the code
(2) Readability/understandability of the code
(3) Ease of modification

### 1. Efficiency

The efficiency of the pretty printer must not only look at the pretty printer code but also the Kodiyak compiler. The Kodiyak compiler was not written to be optimized. With the limited time that was used to design and write the code it is really amazing that it is as fast as it is. Figure 4.4 shows some statistics for file length compared with time to format and print the reformatted file.

| FILE LENGTH (bytes) | TIME (seconds) |
|---|---|
| 70 | 1.4 |
| 151 | 2.2 |
| 457 | 5.3 |
| 819 | 8.8 |
| 1594 | 18.1 |
| 2706 | 32.7 |
| 4275 | 46.8 |
| 5428 | 62.0 |

The time should be considered in relative terms and not absolute values

Figure 4.4 Pretty Printer Statistics

Along the same lines the pretty printer was not written in any optimized form. The function SPACES is not efficiently coded. It is efficient in time but not in space. The most important point is that although it may not be efficient it works. Figure 4.5 shows a segment of the SPACES function. If efficiency is an important issue this function can be modified to improve its efficiency. Also, as stated earlier, if the line length is increased this function will also have to be modified.

```
xstring vspaces (lenstr)
int lenstr;
{
    int x;
    x = lenstr;
    switch(x)
    {
        case 0 : return (xstring) "";
                 break;
        case 1 : return (xstring) " ";
                 break;
        case 2 : return (xstring) "  ";
                 break;
        case 3 : return (xstring) "   ";
                 break;
                .
                .
                .
        default : return (xstring) "";
                  break;
      }
}
```

Figure 4.5 Spaces

2. Readability

Readability is concerned with the user being able to read the code for the pretty printer and without too much effort understand exactly what is going on. Increasing the code's readability is the fact that there are only six attributes each with a unique function that does not change. Additionally, standard rules are followed and there is a standardization among the implementation of the production rules. This basically says if the user can understand one production rule he can probably understand the majority of them. There are a few special rules (and some exceptions) that may take the user a little longer to understand but overall the pretty printer is fairly straight forward.

Since it has few attributes and only uses simple mathematics and string concatenation the code is fairly simple. Probably the most complex notation used involves the if-then-else and if-then-else_if evaluation rules. The syntax for these constructs are a little different but after reading through a few of them they become straightforward (Figure 2.8 explains the syntax).

Since there is standardization and limited complexity it would appear that the pretty printer code should be fairly easy to read and understand. One drawback is by increasing readability some efficiency has been lost. Since this is a research project readability and understandability of the Kodiyak compiler and the pretty printer are more important than optimization. Increasing optimization often decreases the readability of any code. The pretty printer code is fairly straight forward and very readable.

### 3. Ease of Modification

For any number of reasons the existing code may be changed. Is this change an easy and uncomplicated undertaking and/or is it going to be time consuming? The pretty printer code consists of six attributes and simple mathematics. Each attribute is unique and its function or value does not affect other attributes (except bcursor and ecursor which depend on each other). Therefore changing the meaning of one attribute should be straight forward and easy to implement. To add an additional attribute (similar to the type already implemented) will be time consuming (typing time) but fairly simple. On the other hand to add an attribute that uses higher order functions or depends on the value of existing attribute values will be time consuming and complex (each production rule will have to be looked at individually).

From experience the effort in making changes is time consuming but not very complicated. As an example, a modification was made in the implementation of the production rules (and the children of the production rules) using the symbol "expression". Prior to the change the end of line check did not include the use of the length attribute. This length attribute was added to all production rules very easily. It was time consuming because of the major edit to the pretty printer code and the need to verify all corrections/additions were made. The change itself was a very simple modification.

## D. APPLICATION EXTENSION

It is highly desirable to apply the techniques used in this implementation to the development of a language independent pretty printer. Before a generalization can be reached a careful analysis of the existing code must be completed.

### 1. Pretty Printer Code Analysis

Chapter three covers the exact design and implementation in detail. Specifically four general rules for this pretty printer are explained along with all exceptions to each of the four general rules. This section looks at the overall development of these rules and generalizes the method used to apply to any language.

Examination of the pretty printer production rules, looking at how each production rule symbol is implemented, leads to a list of generalized symbol categories. Each SPEC language symbol uniquely fits into one of these four categories (as listed in Figure 4.6).

An analysis of the production rules of SPEC provides the insight into the development of the four generalized symbol categories listed in Figure 4.6. Look at the production rules as collections of symbol categories instead of individual rules. In other words look at the production rules as combinations of keywords (all three types), terminals (CL and VL) and nonterminals. In this way generalizations can be reached. Keywords (general, expression and special) are listed in Figures 3.3, 3.4 and 3.5. All the terminal symbols of the language are listed in Figures 3.16, 3.17 and 3.18. Review of these figures and the pretty printer code leads to the development of the general symbol categories for the production rules. Figure 4.7 lists the production rules from the SPEC grammar transposed into standard forms using these symbol categories. Note that Figure 4.7 contains keywords (all three types grouped under one heading), CL and VL terminal symbols and nonterminals.

1. Keyword
2. Terminal
3. Comment
4. Nonterminal

Figure 4.6 General Symbol Categories

By referring back to the implementation rules for the pretty printer (Figures 3.15, 3.19, 3.20, 3.23 and

1. KEYWORD
2. CL-terminal
3. VL-terminal
4. nonterminal(s)
5. CL-terminal VL-terminal
6. CL-terminal nonterminal(s)
7. nonterminal(s) VL-terminal
8. VL-terminal nonterminal(s)
9. KEYWORD nonterminal(s)
10. KEYWORD nonterminal(s) KEYWORD nonterminal(s) CL-terminal VL-terminal
11. nonterminal(s) KEYWORD nonterminal(s)
12. KEYWORD VL-terminal nonterminal(s)
13. VL-terminal CL-terminal nonterminal(s)
14. CL-terminal nonterminal(s) CL-terminal(*)
15. nonterminal(s) CL-terminal nonterminal(s)
16. CL-terminal VL-terminal CL-terminal nonterminal(s)
17. CL-terminal nonterminal(s) CL-terminal nonterminal(s)(*)
18. KEYWORD nonterminal(s) KEYWORD nonterminal(s)
19. KEYWORD nonterminal(s) CL-terminal nonterminal(s)
20. nonterminal(s) CL-terminal nonterminal(s) CL-terminal(*)
21. nonterminal(s) CL-terminal nonterminal(s) CL-terminal nonterminal(s) CL-terminal(*)
22. KEYWORD VL-terminal nonterminal(s) CL-terminal nonterminal(s)
23. KEYWORD VL-terminal nonterminal(s) KEYWORD nonterminal(s)
24. CL-terminal nonterminal(s) CL-terminal nonterminal(s) CL-terminal(*)
25. KEYWORD CL-terminal nonterminal(s) CL-terminal nonterminal(s) (*)
26. nonterminal(s) KEYWORD nonterminal(s) KEYWORD nonterminal(s)
27. nonterminal(s) KEYWORD VL-terminal KEYWORD VL-terminal nonterminal(s)
28. KEYWORD nonterminal(s) KEYWORD nonterminals(s) KEYWORD nonterminal(s) KEYWORD
29. nonterminal(s) KEYWORD nonterminal(s) CL-terminal nonterminal(s) KEYWORD nonterminal(s)

(*)VL-terminal symbols are matched pairs (i.e., (,),[,] )

Figure 4.7 Standard Forms

3.24) more generalizations can be made. VL-terminal and CL-terminal are implemented exactly the same way. The distinction between the two symbols exists in the exceptions to the terminal symbol rule. Therefore CL-terminal and VL-terminal symbols can be combined into the category terminal. Figure 4.7 grouped the symbol comment as a nonterminal. Because comment is implemented differently than a nonterminal, comment needs to be in a separate category. None of the remaining symbols can be combined. This leads to the modification of the standard forms into a new list of standard forms as listed in Figures 4.8 and 4.9. It can be concluded that each production rule is a combination of one or more of the four general symbol categories (listed in Figure 4.6) repeated one or more times.

### 2. Language Independent Pretty Printer

The approach used in this particular implementation is very regular and could be applied mechanically by a preprocessor. All the information that the preprocessor obtained would be translated and sent to the Kodiyak compiler. The preprocessor is responsible for the language dependent questions as well as any special features the user wanted considered for their particular implementation. Language dependent questions include such items as file name containing the grammar of the language to be pretty printed, keyword specifications, terminal symbols and

```
 1.  KEYWORD
 2.  terminal(s)
 3.  nonterminal(s)
 4.  nonterminal(s) terminal(s)
 5.  terminal(s) nonterminal(s)
 6.  KEYWORD nonterminal(s)
 7.  KEYWORD nonterminal(s) KEYWORD
 8.  nonterminal(s) KEYWORD nonterminal(s)
 9.  KEYWORD terminal nonterminal(s)
10.  terminal nonterminal(s) terminal(*)
11.  nonterminal(s) terminal nonterminal(s)
12.  terminal nonterminal(s) terminal nonterminal(s) (*)
13.  nonterminal terminal nonterminal(s) terminal (*)
14.  KEYWORD terminal nonterminal(s) KEYWORD
       nonterminal(s)
15.  nonterminal(s) KEYWORD nonterminal(s) KEYWORD
       nonterminal(s)
16.  nonterminal(s) terminal nonterminal(s)
       terminal nonterminal terminal(*)
17.  KEYWORD nonterminal(s) KEYWORD nonterminals(s)
       KEYWORD nonterminal(s) KEYWORD


 (*)terminal symbols are matched pairs (i.e.,  (,),[,])
```

Figure 4.8 Standard Forms Revised Without Comment Symbol

1. nonterminal(s) comment
2. comment nonterminal(s)
3. terminal(s) nonterminal(s) comment
4. KEYWORD nonterminal(s) comment
5. terminal nonterminal(s) terminal comment(*)
6. KEYWORD nonterminal(s) KEYWORD comment
7. nonterminal(s) terminal comment nonterminal
8. KEYWORD nonterminal(s) comment nonterminal(s)
9. nonterminal(s) terminal comment nonterminal(s) comment
10. KEYWORD terminal nonterminal(s) terminal comment (*)
11. KEYWORD nonterminal comment nonterminal(s) comment
12. nonterminal(s) KEYWORD nonterminal(s) KEYWORD comment
13. nonterminal(s) KEYWORD nonterminal comment nonterminal(s) comment
14. KEYWORD nonterminal KEYWORD nonterminal comment nonterminal(s)
15. nonterminal KEYWORD terminal KEYWORD terminal comment
16. KEYWORD nonterminal(s) terminal nonterminal comment nonterminal
17. terminal nonterminal(s) terminal comment nonterminal(s) terminal(*)
18. KEYWORD terminal nonterminal(s) terminal nonterminal comment terminal (*)
19. nonterminal KEYWORD nonterminal(s) comment KEYWORD nonterminal comment
20. nonterminal(s) KEYWORD nonterminal(s) terminal nonterminal(s) comment nonterminal(s) KEYWORD comment

(*)terminal symbols are matched pairs (i.e., (,),[,] )

Figure 4.9 Standard Forms Revised With Comment Symbol

values, keyword types, paired terminal symbols, comment symbol, etc. The grammar file should be formatted as specified by the Kodiyak manual [Ref. 9:pp. 2-25]. Special features would include such things as the standard indentation, width of the paper, unusual lengths of tokens, left-hand margin to start at a value other than one, special handling for a grammar rule, etc.

Figure 4.10 shows how the preprocessor works. The preprocessor could be either menu driven (asking a series of questions) or a command line format could be used (i.e., pp grammarfilename -w 120 where pp invokes the preprocessor, grammarfilename is the name of the file containing the grammar and -w 120 states the line length is 120 characters). This method for invoking the preprocessor would be system dependent. The preprocessor would take in all necessary data, use a set of production rule and generate an attribute grammar, format the attribute grammar to be compatible with the Kodiyak compiler and transmit its data to the Kodiyak compiler which in turn would produce a working pretty printer for the desired language.

The rules used by the preprocessor to generate the attribute grammar to be used by the Kodiyak compiler in generating the language independent pretty printer are very straight forward. Figure 4.11 shows the four rules needed to implement a language independent pretty printer.

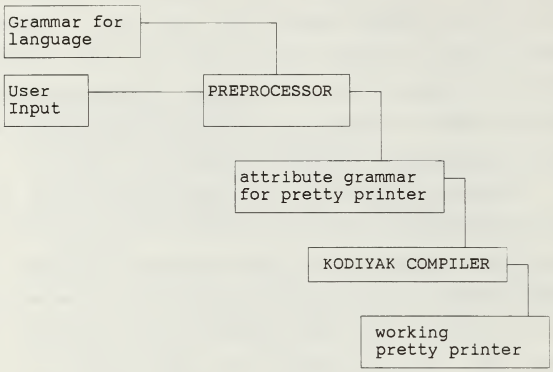

Figure 4.10 Preprocessor

Each production rule, for the given grammar, must be transformed into a set of attribute equations to produce the desired pretty printer code. The four language independent rules listed in Figure 4.11 use the same attributes used in the language dependent pretty printer implementation. Each symbol in a production rule must be categorized and then the associated rule for that symbol must be applied.

As an example consider the production rule "X : A B C D". The preprocessor first would determine the category for each of the four symbols in this production rule. Next

```
1. X : Keyword
    X.str_value = if standard
                    then ["\n",indent,keyword]
                  else if keyword fits on current line
                    then   [keyword]
                  else     ["\n",indent,keyword]
    X.ecursor = if standard
                  then len(indent) + len(Keyword)
                else if keyword fits on current line
                  then   X.bcursor + len(keyword)
                else     len(indent) + len(keyword)

2. X :  Nonterminal
    Nonterminal.bcursor = if (X.bcursor < 0)
                            then len(padding)
                          else  X.bcursor
    X.ecursor = Nonterminal.ecursor
    Nonterminal.padding = X.padding
    Nonterminal.indent = [X.indent, spaces(3)]
    X.str_value = if (X.bcursor < 0)
                    then  [padding,nonterminal.str_value
                  else nonterminal.str_value

3. X :  Terminal
    X.str_value = if (X.bcursor + len(terminal)) < END
                    then terminal
                  else    ["\n",padding,terminal]
    X.ecursor = if (X.bcursor + len(terminal)) < END
                   then  X.bcursor + len(terminal)
                else    len(padding) + len(terminal)
          * if paired terminal symbol check for <= END

     END  = MAXIMUM LINE LENGTH

4. X : Comment
    X.str_value = Comment.str_value
    X.ecursor = if len(Comment.str_value ) > 0
                  then  -1  (*comment exists*)
                else    +1
```

Figure 4.11   Language Independent Rules

the four rules, listed in Figure 4.11, would be applied to each symbol in the production rule. Finally, a set of attribute equations is generated. Figure 4.12 outlines the details of the generation of the attribute equations for this example.

```
PRODUCTION RULE ->     X : A B C D

1.  Assume A is a keyword (standard), B is a nonterminal
          C is a terminal, D is a comment

2.  A.str_value = ["\n",X.indent,Keyword]
    A.ecursor = len(X.indent) + len(Keyword)

3.  B.bcursor = A.ecursor
    Nonterminal.bcursor = if (B.bcursor < 0)
                            then len(padding)
                          else B.bcursor
    Nonterminal.padding = X.padding
    Nonterminal.indent = [X.indent, "   "]
    B.str_value = if (B.bcursor < 0)
                    then [X.padding,Nonterminal.str_value]
                    else Nonterminal.str_value

4.  C.bcursor = B.ecursor
    C.str_value = if (C.bcursor + len(terminal)) < END
                    then terminal
                    else ["\n",X.padding,terminal]
    C.ecursor = if (C.bcursor + len(terminal)) < END
                then C.bcursor + len(terminal)
                else len(X.padding) + len(terminal)

5.  D.str_value = Comment.str_value
    D.ecursor = if len(Comment.str_value) > 0
                then -1
                else +1

6.  X.str_value = [A.str_value,B.str_value,C.str_value,
                     D.str_value]
    X.ecursor = D.ecursor
```

Figure 4.12 Attribute Equation Generation Example

In conclusion it is feasible to use Kodiyak to make a language independent pretty printer generator. In order to do this a preprocessor is needed to gather information on the specific language implementation and any requirements from the user. The preprocessor will take its gathered information, translate it into an attribute grammar for pretty printer (using rules outlined in Figure 4.11) and transmit the attribute grammar to the Kodiyak compiler. The Kodiyak compiler will produce the executable code for a pretty printer for the desired input language. With this method only one pretty printer needs to be written and multiple languages can be pretty printed.

# APPENDIX A

```
| version stamp $Header: spec.k,v 1.5 88/02/16 13:27:58 berzins Exp $

| In the grammar, comments go from a "|" to the end of the line.
| Terminal symbols are entirely upper case or enclosed in single quotes (').
| Nonterminal symbols are entirely lower case.
| Lexical character classes start with a captial letter and are enclosed in {}.
| In a regular expression, x+ means one or more x's.
| In a regular expression, x* means zero or more x's.
| In a regular expression, [xyz] means x or y or z.
| In a regular expression, [^xyz] means any character except x or y or z.
| In a regular expression, [a-z] means any character between a and z.
| In a regular expression, . means any character except newline.

| definitions of lexical classes

%define Digit        :[0-9]
%define Int          :{Digit}+
%define Letter       :[a-zA-Z]
%define Alpha        :({Letter}¦{Digit}¦"_")
%define Blank        :[ \t\n]
%define Quote        :["]
%define Backslash    :"\\"
%define Char         :([^"\\]¦{Backslash}{Quote}¦{Backslash}{Backslash})

| definitions of white space and comments

                    :{Blank}+
                    :"--".*"\n"

| definitions of compound symbols and keywords

AND                 :"&"
OR                  :"¦"
NOT                 :"~"
IMPLIES             :"=>"
IFF                 :"<=>"

LE                  :"<="
GE                  :">="
NE                  :"~="
NLT                 :"~<"
NGT                 :"~>"
NLE                 :"~<="
NGE                 :"~>="
```

```
EQV              :"=="
NEQV             :"~=="

RANGE            :".."
APPEND           :"¦¦"
MOD              :{Backslash}¦MOD
EXP              :"**"

BIND             :"::"
ARROW            :"->"

IF               :IF
THEN             :THEN
ELSE             :ELSE
IN               :IN
U                :U

ALL              :ALL
SOME             :SOME
NUMBER           :NUMBER
SUM              :SUM
PRODUCT          :PRODUCT
SET              :SET
MAXIMUM          :MAXIMUM
MINIMUM          :MINIMUM
UNION            :UNION
INTERSECTION     :INTERSECTION
SUCH             :SUCH{Blank}*THAT
ELSE_IF          :ELSE{Blank}*IF

AS               :AS
CHOOSE           :CHOOSE
CONCEPT          :CONCEPT
DEFINITION       :DEFINITION
DELAY            :DELAY
DO               :DO
END              :END
EXCEPTION        :EXCEPTION
EXPORT           :EXPORT
FI               :FI
FOREACH          :FOREACH
FROM             :FROM
FUNCTION         :FUNCTION
GENERATE         :GENERATE
HIDE             :HIDE
IMPORT           :IMPORT
INHERIT          :INHERIT
INITIALLY        :INITIALLY
INSTANCE         :INSTANCE
INVARIANT        :INVARIANT
ITERATOR         :ITERATOR
```

```
MACHINE             :MACHINE
MESSAGE             :MESSAGE
MODEL               :MODEL
OD                  :OD
OF                  :OF
OPERATOR            :OPERATOR
OTHERWISE           :OTHERWISE
PERIOD              :PERIOD
RENAME              :RENAME
REPLY               :REPLY
SEND                :SEND
STATE               :STATE
TEMPORAL            :TEMPORAL
TIME                :TIME
TO                  :TO
TRANSACTION         :TRANSACTION
TRANSITION          :TRANSITION
TYPE                :TYPE
VALUE               :VALUE
VIRTUAL             :VIRTUAL
WHEN                :WHEN
WHERE               :WHERE

SECONDS             :SECONDS
MINUTES             :MINUTES
HOURS               :HOURS
DAYS                :DAYS
WEEKS               :WEEKS
NANOSEC             :NANOSEC
MICROSEC            :MICROSEC
MILLISEC            :MILLISEC

INTEGER_LITERAL     :{Int}
REAL_LITERAL        :{Int}"."{Int}
CHAR_LITERAL        :"'"."'"
STRING_LITERAL      :{Quote}{Char}*{Quote}

NAME                :{Letter}{Alpha}*

| operator precedences
| %left means 2+3+4 is (2+3)+4.

%left         ';', IF, DO, EXCEPTION, NAME, SEMI;
%left         ',', COMMA;
%left         SUCH;
%left         IFF;
%left         IMPLIES;
%left         OR;
%left         AND;
%left         NOT;
%left         '<', '>', '=', LE, GE, NE, NLT, NGT, NLE, NGE, EQV, NEQV;
```

```
%nonassoc    IN, RANGE;
%left        U, APPEND;
%left        '+', '-', PLUS, MINUS;
%left        '*', '/', MUL, DIV, MOD;
%left        UMINUS;
%left        EXP;
%left        '$', '[', '(', '{', '.', DOT, WHERE;
%left        STAR;

%%
!attribute declarations

%%
! productions of the grammar

start
    : spec
      { }
    ;

spec
    : spec module
      { }
    |
      { }
    ;

      ! A production with nothing after the "|" means the empty string
      ! is a legal replacement for the left hand side.

module
    : function
      { } .
    | machine
      { }
    | type
      { }
    | definition
      { }
    | instance   ! of a generic module
      { }
    ;

function
    : optionally_virtual FUNCTION interface messages concepts END
      { }
    ;
      ! Virtual modules are for inheritance only, never used directly.

machine
```

```
    : optionally_virtual MACHINE interface state messages transactions temporals concepts
END
      { }
    ;

type
    : optionally_virtual TYPE interface model messages transactions temporals concepts END
      { }
    ;

definition
    : DEFINITION interface concepts END
      { }
    ;

instance
    : optionally_virtual INSTANCE parametrized_name '=' parametrized_name hide renames END
      { }
    ;
      | For making instances or partial instantiations of generic modules,
      | and for making interface adjustments to reusable components
      | by hiding or changing some names.

interface
    : NAME formal_parameters inherits imports export
      { }
    ;

      | This part describes the static aspects of a module's interface.
      | The dynamic aspects of the interface are described in the messages.
      | A module is generic iff it has parameters.
      | The parameters can be constrained by a WHERE clause.
      | A module can inherit the behavior of other modules.
      | A module can import concepts from other modules.
      | A module can export concepts for use by other modules.

inherits
    : inherits INHERIT parametrized_name hide renames
      { }
    |
      { }
    ;

      | ancestors are generalizations or simplified views of a module
      | an actor inherits all of the behavior of its ancestors

hide
    : HIDE name_list
      { }
    |
      { }
    ;

      ! Useful for providing limited views of an actor.
      ! Different user classes may see different views of a system.
      ! Messages and concepts can be hidden.

renames
    : renames RENAME NAME AS NAME
      { }
    |
      { }
    ;

      ! Renaming is useful for preventing name conflicts when inheriting
      ! from multiple sources, and for adapting modules for new uses.
      ! The parameters, model and state components, messages, exceptions,
      ! and concepts of an actor can be renamed.

imports
    : imports IMPORT name_list FROM parametrized_name
      { }
    |
      { }
    ;

export
    : EXPORT name_list
      { }
    |
      { }
    ;

messages
    : messages message
      { }
    |
      { }
    ;

message
    : MESSAGE message_header operator response
      { }
    ;

response
    : response_body
      { }
    | response_cases
      { }
    ;
```

```
response_cases
    : WHEN expression_list response_body response_cases
      { }
    | OTHERWISE response_body
      { }
    ;

response_body
    : choose reply sends transition
      { }
    ;

choose
    : CHOOSE '(' field_list restriction ')'
      { }
    |
      { }
    ;
reply
    : REPLY message_header where
      { }
    | GENERATE message_header where       ! used in iterators
      { }
    |
      { }
    ;

sends
    : sends send
      { }
    |
      { }
    ;

send
    : SEND message_header TO parametrized_name where foreach
      { }
    ;

transition
    : TRANSITION expression_list     ! for describing state changes
      { }
    |
      { }
    ;

message_header
    : optional_exception optional_name formal_arguments
      { }
    ;
```

```
where
    : WHERE expression_list
      { }
    |   %prec SEMI       ! must have a lower precedence than WHERE
      { }
    ;

optionally_virtual
    : VIRTUAL
      { }
    |
      { }
    ;

optional_exception
    : EXCEPTION
      { }
    |   %prec SEMI
      { }
    ;

operator
    : operator OPERATOR operator_list
      { }
    |
      { }
    ;

foreach
    : FOREACH '(' field_list restriction ')'
      { }
    |
      { }   .
    ;
      ! FOREACH is used to describe a set of messages to be sent.

concepts
    : concepts concept
      { }
    |
      { }
    ;

concept
    : CONCEPT NAME formal_parameters ':' type_spec where
      ! constants
      { }
    | CONCEPT NAME formal_parameters formal_arguments where VALUE formal_arguments where
      ! functions
      { }
    ;
```

```
model                  ! data types have conceptual models for values
    : MODEL formal_arguments invariant
      { }
    | MODEL formal_arguments invariant initially
      ! Initially clause specifies automatic variable initialization
      { }
    ;

state                  ! machines have conceptual models for states
    : STATE formal_arguments invariant initially
      { }
    ;

invariant              ! invariants are true in all states
    : INVARIANT expression_list
      { }
    ;

initially              ! initial conditions are true only at the beginning
    : INITIALLY expression_list
      { }
    ;

transactions
    : transactions transaction
      { }
    |
      { }
    ;

transaction
    : TRANSACTION parametrized_name '=' action_expression where
      { }
    ;
      ! Transactions are atomic.
      ! The where clause can specify timing constraints.

action_expression
    : action_expression ';' action_list    %prec SEMI    ! sequence
      { }
    | action_list
      { }
    ;

action_list
    : action_list action_list    %prec STAR    ! parallel
      { }
    | IF alternatives FI    ! choice
      { }
    | DO alternatives OD    ! repetition
```

```
        { }
      | parametrized_name      ! a normal message
        { }
      | EXCEPTION parametrized_name    ! an exception message
        { }
      ;

alternatives
      : alternatives OR guard action_expression
        { }
      | guard action_expression
        { }
      ;

guard
      : WHEN expression ARROW
        { }
      |
        { }
      ;

temporals
      : temporals temporal
        { }
      |
        { }
      ;

temporal
      : TEMPORAL NAME where response
        { }
      ;
        ! Temporal events are trigged at absolute times,
        ! in terms of the local clock of the actor.
        ! The "where" describes the triggering conditions
        ! in terms of "TIME" and "PERIOD".

formal_parameters       ! parameter values are determined at specification time
      : '{' field_list '}' where
        { }
      |
        { }
      ;

formal_arguments        ! arguments are evaluated at run-time
      : '(' field_list ')'
        { }
      |
        { }
      ;
```

```
field_list
    : field_list ',' field
      { }
    | field
      { }
    ;

field
    : name_list ':' type_spec
      { }
    | '$' NAME ':' type_spec
      { }
    | '?'
      { }
    ;

type_spec
    : parametrized_name        ! name of a data type
      { }
    | TYPE actual_parameters
      { }
    | FUNCTION actual_parameters
      { }
    | MACHINE actual_parameters
      { }
    | ITERATOR actual_parameters
      { }
    | '?'
      { }
    ;

name_list
    : name_list NAME
      { }
    | NAME
      { }
    ;

optional_name
    : NAME formal_parameters
      { }
    |
      { }
    ;

parametrized_name
    : NAME actual_parameters
      { }
    ;

actual_parameters      ! parameter values are determined at specification time
```

```
    : '{' arg_list '}'
      { }
    |   %prec SEMI        ! must have a lower precedence than '{'
      { }
    ;

actual_arguments        ! arguments are evaluated at run-time
    : '(' arg_list ')'
      { }
    |   %prec SEMI        ! must have a lower precedence than '('
      { }
    ;

arg_list
    : arg_list ',' arg    %prec COMMA
      { }
    | arg
      { }
    ;

arg
    : expression
      { }
    | pair
      { }
    ;

expression_list
    : expression_list ',' expression      %prec COMMA
      { }
    | expression
      { }
    ;

expression
    : quantifier '(' field_list restriction BIND expression ')'
      { }
    | parametrized_name actual_arguments
      { }
    | parametrized_name '@' parametrized_name actual_arguments
      { }
    | NOT expression                %prec NOT
      { }
    | expression AND expression     %prec AND
      { }
    | expression OR expression      %prec OR
      { }
    | expression IMPLIES expression     %prec IMPLIES
      { }
    | expression IFF expression     %prec IFF
      { }
```

```
¦ expression '<' expression     %prec LE
  { }
¦ expression '>' expression     %prec LE
  { }
¦ expression '=' expression     %prec LE
  { }
¦ expression LE expression      %prec LE
  { }
¦ expression GE expression      %prec LE
  { }
¦ expression NE expression      %prec LE
  { }
¦ expression NLT expression     %prec LE
  { }
¦ expression NGT expression     %prec LE
  { }
¦ expression NLE expression     %prec LE
  { }
¦ expression NGE expression     %prec LE
  { }
¦ expression EQV expression     %prec LE
  { }
¦ expression NEQV expression    %prec LE
  { }
¦ '-' expression                %prec UMINUS
  { }
¦ expression '+' expression     %prec PLUS
  { }
¦ expression '-' expression     %prec MINUS
  { }
¦ expression '*' expression     %prec MUL
  { }
¦ expression '/' expression     %prec DIV
  { }
¦ expression MOD expression     %prec MOD
  { }
¦ expression EXP expression     %prec EXP
  { }
¦ expression U expression       %prec U
  { }
¦ expression APPEND expression  %prec APPEND
  { }
¦ expression IN expression      %prec IN
  { }
¦ '*' expression                %prec STAR
  ! *x is the value of x before a transition
  ! x is the value after the transition
  { }
¦ '$' expression                %prec DOT
  ! $x represents a collection of items rather than just one
  ! s1 = {x, $s2} means s1 = union({x}, s2)
```

```
      ! s1 = [x, $s2] means s1 = append([x], s2)
      { }
    | expression RANGE expression   %prec RANGE
      ! x in [a .. b] iff x in {a .. b} iff a <= x <= b
      ! [a .. b] is sorted in increasing order
      { }
    | expression '.' NAME           %prec DOT
      { }
    | expression '[' expression ']' %prec DOT
      { }
    | '(' expression ')'
      { }
    | '(' expression units ')'       ! timing expression
      { }
    | TIME        ! The current local time, used in temporal events
      { }
    | DELAY       ! The time between the triggering event and the response
      { }
    | PERIOD      ! The time between successive events of this type
      { }
    | literal
      { }
    | literal '@' parametrized_name        ! literal with explicit type
      { }
    | '?'         ! An undefined value to be specified later
      { }
    | '!'         ! An undefined and illegal value
      { }
    | IF expression THEN expression middle_cases ELSE expression FI
      { }
    ;

middle_cases·
    : middle_cases ELSE_IF expression THEN expression
      { }
    |
      { }
    ;

quantifier
    : ALL
      { }
    | SOME
      { }
    | NUMBER
      { }
    | SUM
      { }
    | PRODUCT
      { }
    | SET
```

```
        { }
      | MAXIMUM
        { }
      | MINIMUM
        { }
      | UNION
        { }
      | INTERSECTION
        { }
      ;

restriction
      : SUCH expression
        { }
      |
        { }
      ;

literal
      : INTEGER_LITERAL
        { }
      | REAL_LITERAL
        { }
      | CHAR_LITERAL
        { }
      | STRING_LITERAL
        { }
      | '#' NAME            | enumeration type literal
        { }
      | '[' expressions ']'     | sequence literal
        { }
      | '{' expressions '}'     | set literal
        { }
      | '{' expression ';' expressions '}'     | map literal
        { }
      | '[' pair_list ']'        | tuple literal
        { }
      | '{' pair '}'        | one_of literal
        { }
      ;

        | relation literals are sets of tuples

expressions
      : expression_list
        { }
      |
        { }
      ;

pair_list
```

```
    : pair_list ',' pair
      { }
    | NAME pair
      { }
    | pair
      { }
    ;

pair
    : NAME BIND expression
      { }
    ;

units
    : NANOSEC
      { }
    | MICROSEC
      { }
    | MILLISEC
      { }
    | SECONDS
      { }
    | MINUTES
      { }
    | HOURS
      { }
    | DAYS
      { }
    | WEEKS
      { }
    ;

operator_list
    : operator_list operator_symbol
      { }
    | operator_symbol
      { }
    ;

operator_symbol
    : NOT
      { }
    | AND
      { }
    | OR
      { }
    | IMPLIES
      { }
    | IFF
      { }
    | '<'
```

```
   { }
 | '>'
   { }
 | '='
   { }
 | LE
   { }
 | GE
   { }
 | NE
   { }
 | NLT
   { }
 | NGT
   { }
 | NLE
   { }
 | NGE
   { }
 | EQV
   { }
 | NEQV
   { }
 | '+'
   { }
 | '-'
   { }
 | '*'
   { }
 | '/'
   { }
 | MOD
   { }
 | EXP
   { }
 | U
   { }
 | APPEND
   { }
 | IN
   { }
 | RANGE
   { }
 | '.'
   { }
 | '['
   { }
 ;
```

## APPENDIX B

```
!  version stamp $Header: spec.k,v 1.5 88/02/28 13:27:58 berzins Exp $

! In the grammar, comments go from a "!" to the end of the line.
! Terminal symbols are entirely upper case or enclosed in single quotes (').
! Nonterminal symbols are entirely lower case.
! Lexical character classes start with a captial letter and are enclosed in {}.
! In a regular expression, x+ means one or more x's.
! In a regular expression, x* means zero or more x's.
! In a regular expression, [xyz] means x or y or z.
! In a regular expression, [^xyz] means any character except x or y or z.
! In a regular expression, [a-z] means any character between a and z.
! In a regular expression, . means any character except newline.

! definitions of lexical classes

%define Digit        :[0-9]
%define Int          :{Digit}+
%define Letter       :[a-zA-Z]
%define Alpha        :({Letter}|{Digit}|"_")
%define Blank        :[ \t\n]
%define Quote        :["]
%define Backslash    :"\\"
%define Char         :([^"\\]|{Backslash}{Quote}|{Backslash}{Backslash})

! definitions of white space and comments

                 :{Blank}+
COMMENT          :"--".*"\n"

! definitions of compound symbols and keywords

AND              :"&"
OR               :"|"
NOT              :"~"
IMPLIES          :"=>"
IFF              :"<=>"

LE               :"<="
GE               :">="
NE               :"~="
NLT              :"~<"
NGT              :"~>"
NLE              :"~<="
NGE              :"~>="
```

```
EQV             :"=="
NEQV            :"~=="

RANGE           :".."
APPEND          :"||"
MOD             :{Backslash}|MOD
EXP             :"**"

BIND            :"::"
ARROW           :"->"

IF              :IF
THEN            :THEN
ELSE            :ELSE
IN              :IN
U               :U

ALL             :ALL
SOME            :SOME
NUMBER          :NUMBER
SUM             :SUM
PRODUCT         :PRODUCT
SET             :SET
MAXIMUM         :MAXIMUM
MINIMUM         :MINIMUM
UNION           :UNION
INTERSECTION    :INTERSECTION
SUCH            :SUCH{Blank}*THAT
ELSE_IF         :ELSE{Blank}*IF

AS              :AS
CHOOSE          :CHOOSE
CONCEPT         :CONCEPT
DEFINITION      :DEFINITION
DELAY           :DELAY
DO              :DO
END             :END
EXCEPTION       :EXCEPTION
EXPORT          :EXPORT
FI              :FI
FOREACH         :FOREACH
FROM            :FROM
FUNCTION        :FUNCTION
GENERATE        :GENERATE
HIDE            :HIDE
IMPORT          :IMPORT
INHERIT         :INHERIT
INITIALLY       :INITIALLY
INSTANCE        :INSTANCE
INVARIANT       :INVARIANT
ITERATOR        :ITERATOR
```

```
MACHINE           :MACHINE
MESSAGE           :MESSAGE
MODEL             :MODEL
OD                :OD
OF                :OF
OPERATOR          :OPERATOR
OTHERWISE         :OTHERWISE
PERIOD            :PERIOD
RENAME            :RENAME
REPLY             :REPLY
SEND              :SEND
STATE             :STATE
TEMPORAL          :TEMPORAL
TIME              :TIME
TO                :TO
TRANSACTION       :TRANSACTION
TRANSITION        :TRANSITION
TYPE              :TYPE
VALUE             :VALUE
VIRTUAL           :VIRTUAL
WHEN              :WHEN
WHERE             :WHERE

SECONDS           :SECONDS
MINUTES           :MINUTES
HOURS             :HOURS
DAYS              :DAYS
WEEKS             :WEEKS
NANOSEC           :NANOSEC
MICROSEC          :MICROSEC
MILLISEC          :MILLISEC

INTEGER_LITERAL   :{Int}
REAL_LITERAL      :{Int}"."{Int}
CHAR_LITERAL      :"'"."'"
STRING_LITERAL    :{Quote}{Char}*{Quote}

NAME              :{Letter}{Alpha}*

| operator precedences
| %left means 2+3+4 is (2+3)+4.

%left        ';', IF, DO, EXCEPTION, NAME, SEMI;
%left        ',', COMMA;
%left        SUCH;
%left        IFF;
%left        IMPLIES;
%left        OR;
%left        AND;
%left        NOT;
%left        '<', '>', '=', LE, GE, NE, NLT, NGT, NLE, NGE, EQV, NEQV;
```

```
%nonassoc    IN, RANGE;
%left        U, APPEND;
%left        '+', '-', PLUS, MINUS;
%left        '*', '/', MUL, DIV, MOD;
%left        UMINUS;
%left        EXP;
%left        '$', '[', '(', '{', '.', DOT, WHERE;
%left        STAR;
%left        COMMENT;

%%
!attrubute declatations for nonterminal symbols

start {
       str_value: string;
      };

spec {
       indent: string;
       str_value: string;
     };

module {
        indent: string;
        str_value: string;
       };

function {
          indent: string;
          str_value: string;
         };

machine {
         indent: string;
         str_value: string;
        };

type {
       indent: string;
       str_value: string;
     };

definition {
            indent: string;
             str_value: string;
           };

instance {
          indent: string;
          str_value: string;
         };
```

```
interface {
             indent: string;
             str_value: string;
             bcursor: int;
             padding: string;
           };

inherits {
             indent: string;
             str_value: string;
             bcursor: int;
             padding: string;
           };

hide {
        indent: string;
        str_value: string;
        bcursor: int;
        padding: string;
      };

renames {
           indent: string;
           str_value: string;
           bcursor: int;
           padding: string;
         };

imports {
           indent: string;
           str_value: string;
           bcursor: int;
           padding: string;
         };

export {
          indent: string;
          str_value: string;
          bcursor: int;
          padding: string;
        };

messages {
            indent: string;
            str_value: string;
            bcursor: int;
            padding: string;
          };

message {
```

```
          indent: string;
          str_value: string;
          bcursor: int;
          padding: string;
        };

response {
           indent: string;
           str_value: string;
           bcursor: int;
           padding: string;
         };

response_cases {
                 indent: string;
                 str_value: string;
                 bcursor: int;
                 padding: string;
               };

response_body {
                indent: string;
                str_value: string;
                bcursor: int;
                padding: string;
              };

choose {
         indent: string;
         str_value: string;
         bcursor: int;
         padding: string;
       };

reply {
        indent: string;
        str_value: string;
        bcursor: int;
        padding: string;
      };

sends {
        indent: string;
        str_value: string;
        bcursor: int;
        padding: string;
      };

send {
       indent: string;
       str_value: string;
      bcursor: int;
      padding: string;
    };

transition {
            indent: string;
            str_value: string;
            bcursor: int;
            padding: string;
          };

message_header {
                indent: string;
                str_value: string;
                bcursor: int;
                ecursor: int;
                padding: string;
              };

where {
        indent: string;
        str_value: string;
        bcursor: int;
        ecursor: int;
        padding: string;
      };

optionally_virtual {
                    str_value: string;
                    bcursor: int;
                    ecursor: int;
                  };

optional_exception {
                    indent: string;
                    str_value: string;
                    bcursor: int;
                    ecursor: int;
                    padding: string;
                  };

operator {
           indent: string;
           str_value: string;
           bcursor: int;
           ecursor:int;
           padding: string;
         };

foreach {
          indent: string;
          str_value: string;
          bcursor: int;
          padding: string;
        };

concepts {
           indent: string;
           str_value: string;
           bcursor: int;
           padding: string;
         };

concept {
          indent: string;
          str_value: string;
          bcursor: int;
          padding: string;
        };

model {
        indent: string;
        str_value: string;
        bcursor: int;
        padding: string;
      };

state {
        indent: string;
        str_value: string;
        bcursor: int;
        padding: string;
      };

invariant {
            indent: string;
            str_value: string;
            bcursor: int;
            padding: string;
          };

initially {
            indent: string;
            str_value: string;
            bcursor: int;
            padding: string;
          };

transactions {
               indent: string;
               str_value: string;
               bcursor: int;
               padding: string;
             };

transaction {
              indent: string;
              str_value: string;
              bcursor: int;
              padding: string;
            };

action_expression {
                    str_value: string;
                    bcursor: int;
                    ecursor: int;
                    padding: string;
                    length: int;
                  };

action_list {
              str_value: string;
              bcursor: int;
              ecursor: int;
              padding: string;
              length: int;
            };

alternatives {
               str_value: string;
               bcursor: int;
               ecursor: int;
               padding: string;
               length: int;
             .};

guard {
        str_value: string;
        bcursor: int;
        ecursor: int;
        padding: string;
        length: int;
      };

temporals {
            indent: string;
            str_value: string;
            bcursor: int;
            padding: string;
          };

temporal {
           indent: string;
          str_value: string;
          bcursor: int;
          padding: string;
        };

formal_parameters {
                  indent: string;
                  str_value: string;
                  bcursor: int;
                  ecursor: int;
                  padding: string;
                };

formal_arguments {
                 str_value: string;
                 bcursor: int;
                 ecursor: int;
                 padding: string;
               };

field_list {
           str_value: string;
           bcursor: int;
           ecursor: int;
           padding: string;
           length: int;
         };

field {
        str_value: string;
        bcursor: int;
        ecursor: int;
        padding: string;
        length: int;
      };

type_spec {
           str_value: string;
           bcursor: int;
           ecursor: int;
           padding: string;
           length: int;
         };

name_list {
           str_value: string;
           bcursor: int;
           ecursor: int;
           padding: string;
           length: int;
         };
```

```
optional_name {
                indent: string;
                str_value: string;
                bcursor: int;
                ecursor: int;
                padding: string;
                length: int;
              };

parametrized_name {
                    str_value: string;
                    bcursor: int;
                    ecursor: int;
                    padding: string;
                    length: int;
                  };

actual_parameters {
                    str_value: string;
                    bcursor: int;
                    ecursor: int;
                    padding: string;
                    length: int;
                  };

actual_arguments {
                   str_value: string;
                   bcursor: int;
                   ecursor: int;
                   padding: string;
                   length: int;
                 };

arg_list {
           str_value: string;
           bcursor: int;
           ecursor: int;
           padding: string;
           length: int;
         };

arg {
      str_value: string;
      bcursor: int;
      ecursor: int;
      padding: string;
      length: int;
    };

expression_list {
```

```
                str_value: string;
                bcursor: int;
                ecursor: int;
                padding: string;
                length:int;
               };

expression {
            str_value: string;
            bcursor: int;
            ecursor: int;
            padding: string;
            length: int;
          };

middle_cases {
              str_value: string;
              bcursor: int;
              padding: string;
              length: int;
            };

quantifier {
            str_value: string;
            bcursor: int;
            ecursor: int;
            padding: string;
            length: int;
          };

restriction {
             str_value: string;
             bcursor: int;
             ecursor: int;
             padding: string;
             length: int;
           };

literal {
          str_value: string;
          bcursor: int;
          ecursor: int;
          padding: string;
          length: int;
        };

expressions {
             str_value: string;
             bcursor: int;
             ecursor: int;
             padding: string;
```

```
              length: int;
            };

pair_list {
            str_value: string;
            bcursor: int;
            ecursor: int;
            padding: string;
            length: int;
          };

pair {
       str_value: string;
       bcursor: int;
       ecursor: int;
       padding: string;
       length: int;

     };

units {
        str_value: string;
        bcursor: int;
        ecursor: int;
        padding: string;
        length: int;
      };

operator_list {
                str_value: string;
                bcursor: int;
                ecursor: int;
             .  padding: string;
                length: int;
              };

operator_symbol {
                  str_value: string;
                  bcursor: int;
                  ecursor: int;
                  padding: string;
                  length: int;
                };

comment {
          str_value: string;
          bcursor: int;
          length: int;
        };

! attribute declarations for terminal symbols
```

```
INTEGER_LITERAL {
                  %text: string;
                };

REAL_LITERAL {
               %text: string;
             };

CHAR_LITERAL {
               %text: string;
             };

STRING_LITERAL {
                 %text: string;
               };

NAME {
       %text: string;
     };

COMMENT {
          %text: string;
        };


%%
! productions of the grammar

start
    : comment spec
      {
        %output([comment.str_value, spec.str_value]);
        spec.indent = "";
        comment.bcursor = 0;
      }
    ;

spec
    : spec module
      {
        module.indent = spec[1].indent;
        spec[2].indent = spec[1].indent;
        spec[1].str_value = [spec[2].str_value, module.str_value];
      }
    |
      {
        spec.str_value = "";
      }
    ;

      ! A production with nothing after the "|" means the empty string
    ! is a legal replacement for the left hand side.

module
    : function
      {
        function.indent = module.indent;
        module.str_value = function.str_value;
      }
    | machine
      {
        machine.indent = module.indent;
        module.str_value = machine.str_value;
      }
    | type
      {
        type.indent = module.indent;
        module.str_value = type.str_value;
      }
    | definition
      {
        definition.indent = module.indent;
        module.str_value = definition.str_value;
      }
    | instance          ! of a generic module
      {
        instance.indent = module.indent;
        module.str_value = instance.str_value;
      }
    ;

function
    : optionally_virtual FUNCTION interface messages concepts END comment
      {
        comment.bcursor = 0;
        optionally_virtual.bcursor = len(function.indent);
        interface.indent = [function.indent, spaces(3)];
        messages.indent = [function.indent, spaces(3)];
        concepts.indent = [function.indent, spaces(3)];
        interface.bcursor = optionally_virtual.ecursor + 9;
        interface.padding = [spaces(optionally_virtual.ecursor), spaces(9)];
        messages.bcursor = len(messages.indent);
        messages.padding = messages.indent;
        concepts.bcursor = len(concepts.indent);
        concepts.padding = concepts.indent;
        function.str_value = [optionally_virtual.str_value, "FUNCTION",
          interface.str_value, messages.str_value, concepts.str_value, "\n",
          function.indent, "END", comment.str_value, "\n", "\n"];
      }
    ;
      ! Virtual modules are for inheritance only, never used directly.
```

```
machine
    : optionally_virtual MACHINE interface state messages transactions temporals concepts
      END comment
      {
        comment.bcursor = 0;
        optionally_virtual.bcursor = len(machine.indent);
        interface.indent = [machine.indent, spaces(3)];
        state.indent = [machine.indent, spaces(3)];
        messages.indent = [machine.indent, spaces(3)];
        transactions.indent = [machine.indent, spaces(3)];
        temporals.indent = [machine.indent, spaces(3)];
        concepts.indent = [machine.indent, spaces(3)];
        interface.bcursor = optionally_virtual.ecursor + 8;
        interface.padding = [spaces(optionally_virtual.ecursor), spaces(8)];
        state.bcursor = len(state.indent);
        state.padding = state.indent;
        messages.bcursor = len(messages.indent);
        messages.padding = messages.indent;
        transactions.bcursor = len(transactions.indent);
        transactions.padding = transactions.indent;
        temporals.bcursor = len(temporals.indent);
        temporals.padding = temporals.indent;
        concepts.bcursor = len(concepts.indent);
        concepts.padding = concepts.indent;
        machine.str_value = [optionally_virtual.str_value, "MACHINE ",
          interface.str_value, state.str_value, messages.str_value,
          transactions.str_value, temporals.str_value, concepts.str_value, "\n",
          machine.indent, "END", comment.str_value, "\n", "\n"];
      }
    ;

type
    : optionally_virtual TYPE interface model messages transactions temporals concepts
      END comment
      {
        comment.bcursor = 0;
        optionally_virtual.bcursor = len(type.indent);
        interface.indent = [type.indent, spaces(3)];
        model.indent = [type.indent, spaces(3)];
        messages.indent = [type.indent, spaces(3)];
        transactions.indent = [type.indent, spaces(3)];
        temporals.indent = [type.indent, spaces(3)];
        concepts.indent = [type.indent, spaces(3)];
        interface.bcursor = optionally_virtual.ecursor + 5;
        interface.padding = [spaces(optionally_virtual.ecursor), spaces(5)];
        model.bcursor = len(model.indent);
        model.padding = model.indent;
        messages.bcursor = len(messages.indent);
        messages.padding = messages.indent;
        transactions.bcursor = len(transactions.indent);
        transactions.padding = transactions.indent;
```

```
          temporals.bcursor = len(temporals.indent);
          temporals.padding = temporals.indent;
          concepts.bcursor = len(concepts.indent);
          concepts.padding = concepts.indent;
          type.str_value = [optionally_virtual.str_value, "TYPE ", interface.str_value,
            model.str_value, messages.str_value, transactions.str_value,
            temporals.str_value, concepts.str_value, "\n", type.indent, "END",
            comment.str_value, "\n", "\n"];
        }
      ;

definition
      : DEFINITION interface concepts END comment
        {
          comment.bcursor = 0;
          interface.indent = [definition.indent, spaces(3)];
          concepts.indent = [definition.indent, spaces(3)];
          interface.bcursor = len(definition.indent) + 11;
          interface.padding = [definition.indent, spaces(11)];
          concepts.bcursor = len(concepts.indent);
          concepts.padding = concepts.indent;
          definition.str_value = ["\n", definition.indent, "DEFINITION ",
            interface.str_value, concepts.str_value, "\n", "END", comment.str_value, "\n",
            "\n"];
        }
      ;

instance
      : optionally_virtual INSTANCE parametrized_name '=' parametrized_name comment hide
        renames END comment
        {
          optionally_virtual.bcursor = len(instance.indent);
          hide.indent = [instance.indent, spaces(3)];
          renames.indent = [instance.indent, spaces(3)];
          comment[1].bcursor = parametrized_name[2].ecursor;
          comment[2].bcursor = 0;
          parametrized_name[1].bcursor = optionally_virtual.ecursor +
            len(instance.indent) + 9;
          parametrized_name[1].padding = [spaces(optionally_virtual.ecursor), spaces(9)];
          parametrized_name[2].bcursor = parametrized_name[1].ecursor + 3 <= 80
            -> parametrized_name[1].ecursor + 3
            #  len(parametrized_name[1].padding) + 2;
          parametrized_name[2].padding = parametrized_name[1].padding;
          hide.bcursor = len(hide.indent);
          hide.padding = hide.indent;
          renames.bcursor = len(renames.indent);
          renames.padding = renames.indent;
          instance.str_value = parametrized_name[1].ecursor + 3 <= 80
            -> ["\n", instance.indent, optionally_virtual.str_value, "INSTANCE ",
                parametrized_name[1].str_value, " = ", parametrized_name[2].str_value,
                comment[1].str_value, hide.str_value, renames.str_value, "\n",
              instance.indent, "END", comment[2].str_value, "\n", "\n"]
          #  ["\n", instance.indent, optionally_virtual.str_value, "INSTANCE ",
              parametrized_name[1].str_value, "\n", parametrized_name[1].padding, "= ",
              parametrized_name[2].str_value, comment[1].str_value, hide.str_value,
              renames.str_value, "\n", instance.indent, "END", comment[2].str_value, "\n",
              "\n"];
      }
    ;
      | For making instances or partial instantiations of generic modules,
      | and for making interface adjustments to reusable components
      | by hiding or changing some names.

interface
    : NAME formal_parameters comment inherits imports export comment
      {
        formal_parameters.indent = interface.indent;
        inherits.indent = interface.indent;
        imports.indent = interface.indent;
        export.indent = interface.indent;
        comment[1].bcursor = formal_parameters.ecursor > 0
          -> formal_parameters.ecursor
          #  0;
        comment[2].bcursor = 0;
        formal_parameters.bcursor = interface.bcursor + len(NAME.%text);
        formal_parameters.padding = [interface.padding, spaces(len(NAME.%text))];
        inherits.bcursor = len(inherits.indent);
        inherits.padding = inherits.indent;
        imports.bcursor = len(imports.indent);
        imports.padding = imports.indent;
        export.bcursor = len(export.indent);
        export.padding = export.indent;
        interface.str_value = [NAME.%text, formal_parameters.str_value,
          comment[1].str_value, inherits.str_value, imports.str_value, export.str_value,
          comment[2].str_value];
      }
    ;

      | This part describes the static aspects of a module's interface.
      | The dynamic aspects of the interface are described in the messages.
      | A module is generic iff it has parameters.
      | The parameters can be constrained by a WHERE clause.
      | A module can inherit the behavior of other modules.
      | A module can import concepts from other modules.
      | A module can export concepts for use by other modules.

inherits
    : inherits INHERIT parametrized_name comment hide renames comment
      {
        comment[1].bcursor = parametrized_name.ecursor;
        comment[2].bcursor = 0;
        inherits[2].indent = inherits[1].indent;
        hide.indent = [inherits[1].indent, spaces(3)];
        renames.indent = [inherits[1].indent, spaces(3)];
        parametrized_name.bcursor = len(inherits.indent) + 8;
        parametrized_name.padding = [inherits.indent, spaces(8)];
        hide.bcursor = len(hide.indent);
        hide.padding = hide.indent;
        renames.bcursor = len(renames.indent);
        renames.padding = renames.indent;
        inherits[1].str_value = [inherits[2].str_value, "\n", inherits[1].indent,
          "INHERIT ", parametrized_name.str_value, comment[1].str_value, hide.str_value,
          renames.str_value, comment[2].str_value];
      }
    |
      {
        inherits.str_value = "";
      }
    ;

      ! ancestors are generalizations or simplified views of a module
      ! an actor inherits all of the behavior of its ancestors

hide
    : HIDE name_list comment
      {
        name_list.bcursor = hide.bcursor + 5;
        name_list.padding = [hide.padding, spaces(5)];
        comment.bcursor = name_list.ecursor;
        hide.str_value = ["\n", hide.indent, "HIDE ", name_list.str_value,
          comment.str_value];
      }
    |
      {
        hide.str_value = "";
      }
    ;

      ! Useful for providing limited views of an actor.
      ! Different user classes may see different views of a system.
      ! Messages and concepts can be hidden.

renames
    : renames RENAME NAME AS NAME comment
      {
        renames[2].indent = renames[1].indent;
        renames[2].bcursor = renames[1].bcursor;
        renames[2].padding = renames[1].padding;
        comment.bcursor = (len(renames[1].indent) + 7 + len(NAME[1].%text) <= 80)
            && (len(renames[1].indent) + 7 + len(NAME[1].%text) + 4 +
               len(NAME[2].%text) > 80)
          -> len(renames[1].padding) + 6 + len(NAME[2].%text)
          #  len(renames[1].indent) + 11 + len(NAME[1].%text) + len(NAME[2].%text);
        renames[1].str_value = (len(renames[1].indent) + 7 + len(NAME[1].%text) <= 80)
            && (len(renames[1].indent) + 7 + len(NAME[1].%text) + 4 +
               len(NAME[2].%text) > 80)
          -> [renames[2].str_value, "\n", renames[1].indent, "RENAME ", NAME[1].%text,
              "\n", renames[1].padding, spaces(3), "AS ", NAME[2].%text,
              comment.str_value]
          # [renames[2].str_value, "\n", renames[1].indent, "RENAME ", NAME[1].%text,
              " AS ", NAME[2].%text, comment.str_value];
      }
    |
      {
        renames.str_value = "";
      }
    ;

      ! Renaming is useful for preventing name conflicts when inheriting
      ! from multiple sources, and for adapting modules for new uses.
      ! The parameters, model and state components, messages, exceptions,
      ! and concepts of an actor can be renamed.

imports
    : imports IMPORT name_list comment FROM parametrized_name comment
      {
        imports[2].indent = imports[1].indent;
        comment[2].bcursor = parametrized_name.ecursor;
        comment[1].bcursor = name_list.ecursor;
        imports[2].bcursor = imports[1].bcursor;
        imports[2].padding = imports[1].padding;
        name_list.bcursor = len(imports[1].padding) + 7;
        name_list.padding = [imports[1].padding, spaces(7)];
        parametrized_name.bcursor = imports.bcursor + 8;
        parametrized_name.padding = [imports[1].padding, spaces(8)];
        imports[1].str_value = [imports[2].str_value, "\n", imports[1].indent,
          "IMPORT ", name_list.str_value, comment[1].str_value, "\n", imports[1].indent,
          spaces(3), "FROM ", parametrized_name.str_value, comment[2].str_value];
      }
    |
      {
        imports.str_value = "";
      }
    ;

export
    : EXPORT name_list comment
      {
        name_list.bcursor = export.bcursor + 7;
        comment.bcursor = name_list.ecursor;
        name_list.padding = [export.padding, spaces(7)];
        export.str_value = ["\n", export.indent, "EXPORT", name_list.str_value,
          comment.str_value];
      }
    |
      {
        export.str_value = "";
      }
    ;

messages
    : messages message
      {
        messages[2].indent = messages[1].indent;
        message.indent = messages[1].indent;
        messages[2].bcursor = messages[1].bcursor;
        messages[2].padding = messages[1].padding;
        message.bcursor = messages[1].bcursor;
        message.padding = messages[1].padding;
        messages[1].str_value = [messages[2].str_value, message.str_value];
      }
    |
      {
        messages.str_value = "";
      }
    ;

message
    : MESSAGE message_header operator comment response
      {
        message_header.indent = [message.indent, spaces(3)];
        operator.indent = [message.indent, spaces(3)];
        response.indent = [message.indent, spaces(3)];
        comment.bcursor = operator.ecursor;
        message_header.bcursor = message.bcursor + 8;
        message_header.padding = [message.padding, spaces(8)];
        operator.bcursor = message_header.ecursor > 0
          -> message_header.ecursor
          #  message_header.bcursor;
        operator.padding = spaces(operator.bcursor);
        response.bcursor = len(response.indent);
        response.padding = response.indent;
        message.str_value = ["\n", "\n", message.indent, "MESSAGE ",
          message_header.str_value, operator.str_value, comment.str_value,
          response.str_value];
      }
    ;

response
    : response_body
      {
        response_body.indent = response.indent;
        response_body.bcursor = response.bcursor;
        response_body.padding = response.padding;
        response.str_value = response_body.str_value;
        }
      | response_cases
        {
          response_cases.indent = response.indent;
          response_cases.bcursor = response.bcursor;
          response_cases.padding = response.padding;
          response.str_value = response_cases.str_value;
        }
      ;

response_cases
      : WHEN expression_list comment response_body response_cases comment
        {
          response_body.indent = [response_cases[1].indent, spaces(3)];
          response_cases[2].indent = response_cases[1].indent;
          comment[1].bcursor = expression_list.ecursor;
          comment[2].bcursor = 0;
          expression_list.bcursor = response_cases[1].bcursor + 5;
          expression_list.padding = [response_cases[1].padding, spaces(5)];
          response_body.bcursor = response_cases[1].bcursor;
          response_body.padding = response_cases[1].padding;
          response_cases[2].bcursor = response_cases[1].bcursor;
          response_cases[2].padding = response_cases[1].padding;
          response_cases[1].str_value = ["\n", response_cases[1].indent, "WHEN ",
            expression_list.str_value, comment[1].str_value, response_body.str_value,
            response_cases[2].str_value, comment[2].str_value];
          }
      | OTHERWISE response_body
        {
          response_body.indent = [response_cases.indent, spaces(3)];
          response_body.bcursor = response_cases.bcursor + 10;
          response_body.padding = [response_cases.padding, spaces(10)];
          response_cases.str_value = ["\n", response_cases.indent, "OTHERWISE ",
            response_body.str_value];
          }
      ;

response_body
      : choose reply sends transition comment
        {
          comment.bcursor = 0;
          choose.indent = response_body.indent;
          reply.indent = response_body.indent;
          sends.indent = response_body.indent;
          transition.indent = response_body.indent;
          choose.bcursor = len(choose.indent);
          choose.padding = choose.indent;
          reply.bcursor = len(reply.indent);
          reply.padding = reply.indent;
          sends.bcursor = len(sends.indent);
          sends.padding = sends.indent;
      transition.bcursor = len(transition.indent);
      transition.padding = transition.indent;
      response_body.str_value = [choose.str_value, reply.str_value, sends.str_value,
        transition.str_value,comment.str_value];
    }
  ;

choose
  : CHOOSE '(' field_list restriction ')' comment
    {
      field_list.bcursor = choose.bcursor + 7 < 80
        -> choose.bcursor + 7
        #  choose.bcursor + 3;
      comment.bcursor = restriction.ecursor + 1;
      field_list.padding = choose.bcursor + 7 < 80
        -> [choose.padding, spaces(7)]
        #  [choose.padding, spaces(3)];
      restriction.bcursor = field_list.ecursor + restriction.length + 1 <= 80
        -> field_list.ecursor
        #  len(field_list.padding);
      restriction.padding = choose.bcursor + 7 < 80
        -> [choose.padding, spaces(7)]
        #  [choose.padding, spaces(3)];
      choose.str_value = (choose.bcursor + 7 < 80) && (restriction.length == 0)
        -> ["\n", choose.indent, "CHOOSE(", field_list.str_value, restriction.str_value,
            ")", comment.str_value]
        #  (choose.bcursor + 7 >= 80) && (restriction.length == 0)
        -> ["\n", choose.indent, "CHOOSE","\n", spaces(choose.bcursor), spaces(3), "(",
            field_list.str_value, restriction.str_value, ")", comment.str_value]
        #  (choose.bcursor + 7 < 80) && (field_list.ecursor +
           restriction.length + 1 <= 80)
        -> ["\n", choose.indent, "CHOOSE(", field_list.str_value," ",
            restriction.str_value, ")", comment.str_value]
        #  (choose.bcursor + 7 >= 80) && (field_list.ecursor +
           restriction.length + 1 <= 80)
        -> ["\n", choose.indent, "CHOOSE","\n", spaces(choose.bcursor), spaces(3), "(",
            field_list.str_value, " ", restriction.str_value, ")", comment.str_value]
        #  (choose.bcursor + 7 < 80) && (field_list.ecursor +
           restriction.length + 1 > 80)
        -> ["\n", choose.indent, "CHOOSE(", field_list.str_value,"\n",
            field_list.padding, restriction.str_value, ")", comment.str_value]
        #  ["\n", choose.indent, "CHOOSE","\n", spaces(choose.bcursor), spaces(3),
            "(", field_list.str_value,"\n", field_list.padding, restriction.str_value,
            ")", comment.str_value];
    }
  |
    {
      choose.str_value = "";
    }
  ;
```

```
reply
    : REPLY message_header comment where
      {
        comment.bcursor = message_header.ecursor > 0
          -> message_header.ecursor
          #  0;
        message_header.indent = reply.indent;
        where.indent = [reply.indent, spaces(3)];
        message_header.bcursor = reply.bcursor + 6;
        message_header.padding = [reply.padding, spaces(6)];
        where.bcursor = len(where.indent);
        where.padding = where.indent;
        reply.str_value = ["\n", reply.indent, "REPLY ", message_header.str_value,
          comment.str_value, where.str_value];
      }
    | GENERATE message_header comment where       ! used in iterators
      {
        comment.bcursor = message_header.ecursor > 0
          -> message_header.ecursor
          #  0;
        message_header.indent = reply.indent;
        where.indent = [reply.indent, spaces(3)];
        message_header.bcursor = reply.bcursor + 9;
        message_header.padding = [reply.padding, spaces(9)];
        where.bcursor = len(where.indent);
        where.padding = where.indent;
        reply.str_value = ["\n", reply.indent, "GENERATE ", message_header.str_value,
          comment.str_value, where.str_value];
      }
    |
      {
        reply.str_value = "";
      }
    ;

sends
    : sends send
      {
        sends[2].indent = sends[1].indent;
        send.indent = sends[1].indent;
        sends[2].bcursor = sends[1].bcursor;
        sends[2].padding = sends[1].padding;
        send.bcursor = sends[1].bcursor;
        send.padding = sends[1].padding;
        sends[1].str_value = [sends[2].str_value, send.str_value];
      }
    |
      {
        sends.str_value = "";
      }
    ;
```

```
send
    : SEND message_header TO parametrized_name comment where foreach
      {
        comment.bcursor = parametrized_name.ecursor;
        message_header.indent = [send.indent, spaces(3)];
        where.indent = [send.indent, spaces(3)];
        foreach.indent = [send.indent, spaces(3)];
        message_header.bcursor = send.bcursor + 5;
        message_header.padding = [send.padding, spaces(5)];
        parametrized_name.bcursor = send.bcursor + 3;
        parametrized_name.padding = [send.padding, spaces(6)];
        where.bcursor = len(where.indent);
        where.padding = where.indent;
        foreach.bcursor = len(foreach.indent);
        foreach.padding = foreach.indent;
        send.str_value = ["\n", send.indent, "SEND ", message_header.str_value, "\n",
          send.indent, spaces(3), "TO ", parametrized_name.str_value, comment.str_value,
          where.str_value,  foreach.str_value];
      }
    ;

transition
    : TRANSITION expression_list comment    | for describing state changes
      {
        expression_list.bcursor = transition.bcursor + 11;
        comment.bcursor = expression_list.ecursor;
        expression_list.padding = [transition.padding, spaces(11)];
        transition.str_value = ["\n", transition.indent, "TRANSITION ",
          expression_list.str_value, comment.str_value];
      }
    |
      {
        transition.str_value = "";
      }
    ;

message_header
    : optional_exception optional_name formal_arguments
      {
        optional_name.indent = message_header.indent;
        optional_exception.indent = message_header.padding;
        optional_exception.bcursor = message_header.bcursor;
        optional_exception.padding = message_header.padding;
        optional_name.bcursor = optional_exception.ecursor;
        optional_name.padding = spaces(optional_exception.ecursor);
        formal_arguments.bcursor = optional_name.ecursor < 0
          -> optional_name.bcursor
          #  optional_name.length == 0
          -> optional_name.ecursor
          #  len(optional_name.indent);
        formal_arguments.padding = spaces(formal_arguments.bcursor);
        message_header.ecursor = formal_arguments.ecursor;
        message_header.str_value = optional_name.ecursor < 0
          -> [optional_exception.str_value, optional_name.str_value, optional_name.indent,
              formal_arguments.str_value]
          #  optional_name.length == 0
          -> [optional_exception.str_value, optional_name.str_value,
              formal_arguments.str_value]
          #  [optional_exception.str_value, optional_name.str_value, "\n",
              optional_name.indent,formal_arguments.str_value];
      }
    ;

where
    : WHERE expression_list comment
      {
        comment.bcursor = expression_list.ecursor;
        expression_list.bcursor = len(where.indent) + 6;
        expression_list.padding = [where.indent, spaces(6)];
        where.ecursor = comment.length > 0
          -> -1
          #  expression_list.ecursor;
        where.str_value = ["\n",  where.indent, "WHERE ", expression_list.str_value,
          comment.str_value];
      }
    |       %prec SEMI              I must have a lower precedence than WHERE
      {
        where.ecursor = where.bcursor;
        where.str_value = "";
      }
    ;

optionally_virtual
    : VIRTUAL
      {
        optionally_virtual.ecursor = optionally_virtual.bcursor + 8;
        optionally_virtual.str_value = ["\n", "VIRTUAL "];
      }
    |
      {
        optionally_virtual.ecursor = optionally_virtual.bcursor + 0;
        optionally_virtual.str_value = "";
      }
    ;

optional_exception
    : EXCEPTION
      {
        optional_exception.ecursor = optional_exception.bcursor + 10;
        optional_exception.str_value =  "EXCEPTION ";
      }
    |       %prec SEMI
      {
        optional_exception.ecursor = optional_exception.bcursor;
        optional_exception.str_value = "";
      }
    ;

operator
    : operator OPERATOR operator_list
      {
        operator[2].bcursor = operator[1].bcursor;
        operator[2].padding = operator[1].padding;
        operator_list.bcursor = operator[2].ecursor + 10 + operator_list.length <= 80
          -> operator[2].ecursor + 10
          #  len(operator[1].padding) + 9;
        operator_list.padding = spaces(operator_list.bcursor);
        operator[1].ecursor = operator_list.ecursor;
        operator[1].str_value = operator[2].ecursor + 10 + operator_list.length <= 80
          ->  [operator[2].str_value, " OPERATOR ", operator_list.str_value]
          #   [operator[2].str_value, "\n", operator[1].padding,  "OPERATOR ",
               operator_list.str_value];
      }
    |
      {
        operator.ecursor = operator.bcursor;
        operator.str_value = "";
      }
    ;

foreach
    : FOREACH '(' field_list restriction ')' comment
      {
        field_list.bcursor = foreach.bcursor + 8 < 80
          -> foreach.bcursor + 8
          #  len(foreach.indent) + 4;
        field_list.padding = [foreach.padding, spaces(8)];
        restriction.bcursor = field_list.ecursor;
        restriction.padding = field_list.padding;
        comment.bcursor = restriction.ecursor + 1;
        foreach.str_value = (foreach.bcursor + 8 < 80) && (restriction.length == 0)
          -> ["\n", foreach.indent, "FOREACH(", field_list.str_value,
              restriction.str_value, ")", comment.str_value]
          #  (foreach.bcursor + 8 >= 80) && (restriction.length == 0)
          -> ["\n", foreach.indent, "FOREACH", "\n", foreach.indent, spaces(3), "(",
              field_list.str_value, restriction.str_value, ")", comment.str_value]
          #  (foreach.bcursor + 8 < 80) && (restriction.length > 0)
          -> ["\n", foreach.indent, "FOREACH(", field_list.str_value, " ",
              restriction.str_value, ")", comment.str_value]
          #  ["\n", foreach.indent, "FOREACH", "\n", foreach.indent, spaces(3), "(",
              field_list.str_value, " ", restriction.str_value, ")", comment.str_value];
      }
      |
        {
          foreach.str_value = "";
        }
      ;
        ! FOREACH is used to describe a set of messages to be sent.

concepts
      : concepts concept
        {
          concept.indent = concepts[1].indent;
          concepts[2].indent = concepts[1].indent;
          concepts[2].bcursor = concepts[1].bcursor;
          concepts[2].padding = concepts[1].padding;
          concept.bcursor = concepts[1].bcursor;
          concept.padding = concepts[1].padding;
          concepts[1].str_value = [concepts[2].str_value, concept.str_value];
        }
      |
        {
          concepts.str_value = "";
        }
      ;

concept
      : CONCEPT NAME formal_parameters ':' type_spec comment where
        | constants
        {
          comment.bcursor = type_spec.ecursor;
          formal_parameters.indent = [concept.indent, spaces(3)];
          where.indent = [concept.indent, spaces(3)];
          formal_parameters.bcursor = concept.bcursor + 8 + len(NAME.%text);
          formal_parameters.padding = [concept.padding, spaces(8), spaces(len(NAME.%text))];
          type_spec.bcursor = formal_parameters.ecursor < 0
            -> len(formal_parameters.padding) + 2
            #  formal_parameters.ecursor + 2 <= 80
            -> formal_parameters.ecursor + 2
            #  len(formal_parameters.padding) + 2;
          type_spec.padding = formal_parameters.padding;
          where.bcursor = len(where.indent);
          where.padding = where.indent;
          concept.str_value = formal_parameters.ecursor < 0
            -> ["\n", "\n", concept.indent, "CONCEPT ", NAME.%text,
                formal_parameters.str_value, "\n", formal_parameters.padding, ": ",
                type_spec.str_value, comment.str_value, where.str_value]
            #  formal_parameters.ecursor + 2 <= 80
            -> ["\n", "\n", concept.indent, "CONCEPT ", NAME.%text,
                formal_parameters.str_value, ": ", type_spec.str_value, comment.str_value,
                where.str_value]
            #  ["\n", "\n", concept.indent, "CONCEPT ", NAME.%text,
                formal_parameters.str_value, "\n", formal_parameters.padding, ": ",
          type_spec.str_value, comment.str_value, where.str_value];
      }
    | CONCEPT NAME formal_parameters formal_arguments where VALUE formal_arguments where
      ! functions
      {
        formal_parameters.indent = [concept.indent, spaces(3)];
        where[1].indent = [concept.indent, spaces(3)];
        where[2].indent = [concept.indent, spaces(3)];
        formal_parameters.bcursor = concept.bcursor + 8 + len(NAME.%text);
        formal_parameters.padding = [concept.padding, spaces(8), spaces(len(NAME.%text))];
        formal_arguments[1].bcursor = formal_parameters.ecursor > 0
          -> formal_parameters.ecursor
          #  len(formal_parameters.padding);
        formal_arguments[1].padding = formal_parameters.padding;
        where[1].bcursor = len(where[1].indent);
        where[1].padding = where[1].indent;
        formal_arguments[2].bcursor = len(concept.padding) + 3 + 6;
        formal_arguments[2].padding = [concept.padding, spaces(3), spaces(6)];
        where[2].bcursor = len(where[2].indent);
        where[2].padding = where[2].indent;
        concept.str_value = ["\n", "\n", concept.indent, "CONCEPT ", NAME.%text,
          formal_parameters.str_value,  formal_arguments[1].str_value, where[1].str_value,
          "\n", concept.indent, spaces(3), "VALUE ", formal_arguments[2].str_value,
          where[2].str_value];
      }
    ;

model       ! data types have conceptual models for values
    : MODEL formal_arguments invariant
      {
        invariant.indent = [model.indent, spaces(3)];
        formal_arguments.bcursor = len(model.indent) + 6;
        formal_arguments.padding = [model.indent, spaces(6)];
        invariant.bcursor = len(invariant.indent);
        invariant.padding = invariant.indent;
        model.str_value = ["\n", "\n", model.indent, "MODEL ", formal_arguments.str_value,
          invariant.str_value];
      }
    | MODEL formal_arguments invariant initially
      ! initially clause specifies automatic variable initialization
      {
        invariant.indent = [model.indent, spaces(3)];
        initially.indent = [model.indent, spaces(3)];
        formal_arguments.bcursor = len(model.indent) + 6;
        formal_arguments.padding = [model.indent, spaces(6)];
        invariant.bcursor = len(invariant.indent);
        invariant.padding = invariant.indent;
        initially.bcursor = len(initially.indent);
        initially.padding = initially.indent;
        model.str_value = ["\n", "\n",  model.indent, "MODEL ",
          formal_arguments.str_value, invariant.str_value, initially.str_value];
      }
    ;

state        ! machines have conceptual models for states
    : STATE formal_arguments invariant initially
      {
        invariant.indent = [state.indent, spaces(3)];
        initially.indent = [state.indent, spaces(3)];
        formal_arguments.bcursor = len(state.indent) + 6;
        formal_arguments.padding = [state.indent, spaces(6)];
        invariant.bcursor = len(invariant.indent);
        invariant.padding = invariant.indent;
        initially.bcursor = len(initially.indent);
        initially.padding = initially.indent;
        state.str_value = ["\n", "\n", state.indent, "STATE ", formal_arguments.str_value,
          invariant.str_value, initially.str_value];
      }
    ;

invariant    ! invariants are true in all states
    : INVARIANT expression_list comment
      {
        expression_list.bcursor = invariant.bcursor + 10;
        comment.bcursor = expression_list.ecursor;
        expression_list.padding = [invariant.padding, spaces(10)];
        invariant.str_value = ["\n", invariant.indent, "INVARIANT",
          expression_list.str_value, comment.str_value];
      }
    ;

initially    ! initial conditions are true only at the beginning
    : INITIALLY expression_list comment
      {
        expression_list.bcursor = initially.bcursor + 10;
        comment.bcursor = expression_list.ecursor;
        expression_list.padding = [initially.padding, spaces(10)];
        initially.str_value = ["\n", initially.indent, "INITIALLY ",
          expression_list.str_value, comment.str_value];
      }
    ;

transactions
    : transactions transaction
      {
        transactions[2].indent = transactions[1].indent;
        transaction.indent = transactions[1].indent;
        transactions[2].bcursor = transactions[1].bcursor;
        transactions[2].padding = transactions[1].padding;
        transaction.bcursor = len(transaction.indent);
        transaction.padding = transaction.indent;
        transactions[1].str_value = [transactions[2].str_value, transaction.str_value];
      }
    |
      {
        transactions.str_value = "";
      }
    ;

transaction
    : TRANSACTION parametrized_name '=' action_expression comment where
      {
        comment.bcursor = action_expression.ecursor;
        where.indent = [transaction.indent, spaces(3)];
        parametrized_name.bcursor = len(transaction.indent) + 12;
        parametrized_name.padding = [transaction.indent, spaces(12)];
        action_expression.bcursor = parametrized_name.ecursor + 3 <= 80
          -> parametrized_name.ecursor + 3
          #  len(parametrized_name.padding);
        action_expression.padding = parametrized_name.padding;
        where.bcursor = len(where.indent);
        where.padding = where.indent;
        transaction.str_value = parametrized_name.ecursor + 3 <= 80
          -> ["\n", "\n",  transaction.indent, "TRANSACTION ",
              parametrized_name.str_value, " = ", action_expression.str_value,
              comment.str_value, where.str_value]
          #  ["\n", "\n",  transaction.indent, "TRANSACTION ",
              parametrized_name.str_value, " =", "\n", parametrized_name.padding,
              action_expression.str_value, comment.str_value, where.str_value];
      }
    ;
      ! Transactions are atomic.
      ! The where clause can specify timing constraints.

action_expression
    : action_expression ';' comment action_list     %prec SEMI          !sequence
      {
        action_expression[1].length = action_expression[2].length + 2 + comment.length
          + action_list.length;
        comment.bcursor = action_expression[2].ecursor + 2;
        action_expression[2].bcursor = action_expression[1].bcursor;
        action_expression[2].padding = action_expression[1].padding;
        action_list.bcursor = comment.length > 0
          -> len(action_expression[1].padding)
          #  action_expression[2].ecursor + 2 <= 80
          -> action_expression[2].ecursor + 2
          #  len(action_expression[1].padding);
        action_list.padding = action_expression[2].padding;
        action_expression[1].ecursor = action_list.ecursor;
        action_expression[1].str_value = comment.length > 0
          -> [action_expression[2].str_value, ";", comment.str_value,
              action_list.str_value]
          #  action_expression[2].ecursor + 2 <= 80
        -> [action_expression[2].str_value, "; ", comment.str_value,
            action_list.str_value]
        #  [action_expression[2].str_value, ";", "\n", comment.str_value,
             action_expression[1].padding, action_list.str_value];
      }
    | action_list
      {
        action_expression.length = action_list.length;
        action_list.bcursor = action_expression.bcursor;
        action_list.padding = action_expression.padding;
        action_expression.ecursor = action_list.ecursor;
        action_expression.str_value = action_list.str_value;
      }
    ;

action_list
    : action_list action_list        %prec STAR      | parallel
      {
        action_list[1].length = action_list[2].length + action_list[3].length;
        action_list[2].bcursor = action_list[1].bcursor;
        action_list[2].padding = action_list[1].padding;
        action_list[3].bcursor = action_list[2].ecursor;
        action_list[3].padding = action_list[1].padding;
        action_list[1].ecursor = action_list[3].ecursor;
        action_list[1].str_value = [action_list[2].str_value, action_list[3].str_value];
      }
    | IF alternatives FI             | choice
      {
        action_list.length = 3 + alternatives.length + 3;
        alternatives.bcursor = action_list.bcursor + 3 + alternatives.length <= 80
          -> action_list.bcursor + 3
          #  len(action_list.padding) + 3;
        alternatives.padding = action_list.bcursor + 3 + alternatives.length <= 80
          -> [action_list.padding, spaces(3)]
          #  action_list.padding;
        action_list.ecursor = alternatives.ecursor + 3 <= 80
          -> alternatives.ecursor + 3
          #  len(alternatives.padding) + 2;
        action_list.str_value = (action_list.bcursor + 3 + alternatives.length <= 80)
            &&  (alternatives.ecursor + 3 <= 80)
          -> ["IF ", alternatives.str_value, " FI"]
          #  (action_list.bcursor + 3 + alternatives.length > 80)
            &&  (alternatives.ecursor + 3 <= 80)
          -> ["\n", action_list.padding, "IF ", alternatives.str_value, " FI"]
          #  (action_list.bcursor + 3 + alternatives.length <= 80)
            &&  (alternatives.ecursor + 3 > 80)
          -> ["IF ", alternatives.str_value,"\n", alternatives.padding, "FI"]
          #  ["\n", action_list.padding, "IF ", alternatives.str_value, "\n",
              alternatives.padding, "FI"];
      }
    | DO alternatives OD             | repetition
```

```
        {
          action_list.length = 3 + alternatives.length + 3;
          alternatives.bcursor = action_list.bcursor + 3 + alternatives.length <= 80
            -> action_list.bcursor + 3
            #  len(action_list.padding) + 3;
          alternatives.padding = action_list.bcursor + 3 + alternatives.length <= 80
            -> [action_list.padding, spaces(3)]
            #  action_list.padding;
          action_list.ecursor = alternatives.ecursor + 3 <= 80
            -> alternatives.ecursor + 3
            #  len(alternatives.padding) + 2;
          action_list.str_value = (action_list.bcursor + 3 + alternatives.length <= 80)
              &&  (alternatives.ecursor + 3 <= 80)
            -> ["DO ", alternatives.str_value, " OD"]
            # (action_list.bcursor + 3 + alternatives.length > 80)
              &&  (alternatives.ecursor + 3 <= 80)
            -> ["\n", action_list.padding, "DO ", alternatives.str_value, " OD"]
            #  (action_list.bcursor + 3 + alternatives.length < 80)
              &&  (alternatives.ecursor + 3 > 80)
            -> ["DO ", alternatives.str_value,"\n", alternatives.padding, "OD"]
            #  ["\n", action_list.padding, "DO ", alternatives.str_value, "\n",
                alternatives.padding, "OD"];
        }
      | parametrized_name              | a normal message
        {
          action_list.length = parametrized_name.length;
          parametrized_name.bcursor = action_list.bcursor;
          parametrized_name.padding = action_list.padding;
          action_list.ecursor = parametrized_name.ecursor;
          action_list.str_value = parametrized_name.str_value;
        }
      | EXCEPTION parametrized_name    | an exception message
        {
          action_list.length = 10 + parametrized_name.length;
          parametrized_name.bcursor = action_list.bcursor + 10;
          parametrized_name.padding = [action_list.padding, spaces(10)];
          action_list.ecursor = parametrized_name.ecursor;
          action_list.str_value = ["EXCEPTION ", parametrized_name.str_value];
        }
      ;

alternatives
      : alternatives OR guard action_expression
        {
          alternatives[1].length = alternatives[2].length + 3 + guard.length
            + action_expression.length;
          alternatives[2].bcursor = alternatives[1].bcursor;
          alternatives[2].padding = alternatives[1].padding;
          guard.bcursor = alternatives[2].ecursor + 3 <= 80
            -> alternatives[2].ecursor + 3
            #  len(alternatives[1].padding) + 2;
        guard.padding = alternatives[1].padding;
        action_expression.bcursor = guard.ecursor;
        action_expression.padding = alternatives[1].padding;
        alternatives[1].ecursor = action_expression.ecursor;
        alternatives[1].str_value = alternatives[2].ecursor + 3 <= 80
          -> [alternatives[2].str_value, " ¦ ", guard.str_value,
              action_expression.str_value]
          #  [alternatives[2].str_value, "\n", alternatives[1].padding, "¦ ",
              guard.str_value, action_expression.str_value];
      }
    ¦ guard action_expression
      {
        alternatives.length = guard.length + action_expression.length;
        guard.bcursor = alternatives.bcursor;
        guard.padding = alternatives.padding;
        action_expression.bcursor = guard.ecursor;
        action_expression.padding = alternatives.padding;
        alternatives.ecursor = action_expression.ecursor;
        alternatives.str_value = [guard.str_value, action_expression.str_value];
      }
    ;

guard
    : WHEN expression ARROW
      {
        guard.length = 5 + expression.length + 3;
        expression.padding = spaces(expression.bcursor);
        expression.bcursor = guard.bcursor + 5;
        guard.str_value = expression.ecursor + 3 <= 80
          -> ["WHEN ", expression.str_value, " ->", "\n", guard.padding]
          #  ["WHEN ", expression.str_value, "\n", expression.padding, "->", "\n",
              guard.padding];
        guard.ecursor = expression.ecursor + 3 <= 80
          -> len(guard.padding)
          #  len(guard.padding) + 2;
      }
    ¦
      {
        guard.length = 0;
        guard.ecursor = guard.bcursor;
        guard.str_value = "";
      }
    ;

temporals
    : temporals temporal
      {
        temporals[2].indent = temporals[1].indent;
        temporal.indent = temporals[1].indent;
        temporals[2].bcursor = temporals[1].bcursor;
        temporals[2].padding = temporals[1].padding;
        temporal.bcursor = temporals[1].bcursor;
        temporal.padding = temporals.padding;
        temporals[1].str_value = [temporals[2].str_value,temporal.str_value];
      }
    |
      {
        temporals.str_value = "";
      }
    ;

temporal
    : TEMPORAL NAME where response
      {
        where.indent = [temporal.indent, spaces(3)];
        response.indent = [temporal.indent, spaces(3)];
        where.bcursor = len(where.indent);
        where.padding = where.indent;
        response.bcursor = len(response.indent);
        response.padding = response.indent;
        temporal.str_value = ["\n", "\n", temporal.indent, "TEMPORAL ", NAME.%text,
          where.str_value, response.str_value];
      }
    ;
      | Temporal events are trigged at absolute times,
      | in terms of the local clock of the actor.
      | The "where" describes the triggering conditions
      | in terms of "TIME" and "PERIOD".

formal_parameters            | parameter values are determined at specification time
    : '{' field_list '}' where
      {
        where.indent = formal_parameters.indent;
        field_list.bcursor = formal_parameters.bcursor + 1 < 80
          -> formal_parameters.bcursor + 1
          #  len(formal_parameters.padding) + 1;
        field_list.padding = [formal_parameters.padding, spaces(1)];
        where.bcursor = len(where.indent);
        where.padding = where.indent;
        formal_parameters.ecursor = where.ecursor < 0
          -> where.ecursor
          #  where.ecursor <= where.bcursor
          -> field_list.ecursor + 1
          #  where.ecursor;
        formal_parameters.str_value = formal_parameters.bcursor + 1 < 80
          -> ["{", field_list.str_value, "}", where.str_value]
          #  ["\n", formal_parameters.padding, "{", field_list.str_value, "}",
              where.str_value];
      }
    |
      {
        formal_parameters.ecursor = formal_parameters.bcursor;
        formal_parameters.str_value = "";
      }
    ;

formal_arguments              | arguments are evaluated at run-time
    : '(' field_list ')' comment
      {
        comment.bcursor = field_list.ecursor + 1;
        field_list.bcursor = formal_arguments.bcursor + 1 < 80
          -> formal_arguments.bcursor + 1
          #  len(formal_arguments.padding) + 1;
        field_list.padding = [formal_arguments.padding, spaces(1)];
        formal_arguments.ecursor = comment.length == 0
          -> field_list.ecursor + 1
          # -1;
        formal_arguments.str_value = formal_arguments.bcursor + 1 < 80
          -> ["(", field_list.str_value, ")", comment.str_value]
          #  ["\n", formal_arguments.padding, "(", field_list.str_value, ")",
               comment.str_value];
      }
    |  comment
      {
        comment.bcursor = formal_arguments.bcursor;
        formal_arguments.ecursor = formal_arguments.bcursor;
        formal_arguments.str_value = comment.str_value;
      }
    ;

field_list
    : field_list ',' field
      {
        field_list[1].length = field_list[2].length + 2 + field.length;
        field_list[2].bcursor = field_list[1].bcursor;
        field_list[2].padding = field_list[1].padding;
        field.bcursor = field_list[2].ecursor + 2 <= 80
          -> field_list[2].ecursor + 2
          #  len(field_list[1].padding);
        field.padding = field_list[1].padding;
        field_list[1].ecursor = field.ecursor;
        field_list[1].str_value = field_list[2].ecursor  + 2 <= 80
          -> [field_list[2].str_value, ", ", field.str_value]
          #  [field_list[2].str_value, ",", "\n", field_list[1].padding, field.str_value];
      }
    | field
      {
        field_list.length = field.length;
        field.bcursor = field_list.bcursor;
        field.padding = field_list.padding;
        field_list.ecursor = field.ecursor;
        field_list.str_value = field.str_value;
      }
    ;

field
    : name_list ':' type_spec
      {
        field.length = name_list.length + 2 + type_spec.length;
        name_list.bcursor = field.bcursor;
        name_list.padding = field.padding;
        type_spec.bcursor = name_list.ecursor + 2 + type_spec.length <= 80
          -> name_list.ecursor + 2
          #  len(field.padding) + 2;
        type_spec.padding = field.padding;
        field.ecursor = type_spec.ecursor;
        field.str_value = name_list.ecursor + 2 + type_spec.length <= 80
          -> [name_list.str_value, ": ", type_spec.str_value]
          #  [name_list.str_value, "\n", field.padding, ": ", type_spec.str_value];
      }
    | '$' NAME ':' type_spec
      {
        field.length = 1 + len(NAME.%text) + 3 + type_spec.length;
        type_spec.bcursor = field.bcursor + 3 + len(NAME.%text) + 1
          + type_spec.length  <= 80
          -> field.bcursor + 3 + len(NAME.%text)
          #  len(field.padding) + 3 + len(NAME.%text);
        type_spec.padding = field.padding;
        field.ecursor = type_spec.ecursor;
        field.str_value  = field.bcursor + 3 + len(NAME.%text) + 1
          + type_spec.length <= 80
          -> ["$", NAME.%text, ": ", type_spec.str_value]
          #  ["\n", field.padding, "$", NAME.%text,": ", type_spec.str_value];
      }
    | '?'
      {
        field.length = 1;
        field.ecursor = field.bcursor + 1 <= 80
          -> field.bcursor + 1
          #  len(field.padding) + 1;
        field.str_value =  field.bcursor+ 1  <= 80
          -> "?"
          #  ["\n", field.padding, "?"];
      }
    ;

type_spec
    : parametrized_name          | name of a data type
      {
        type_spec.length = parametrized_name.length;
        parametrized_name.bcursor = type_spec.bcursor;
        parametrized_name.padding = type_spec.padding;
        type_spec.ecursor = parametrized_name.ecursor;
        type_spec.str_value = parametrized_name.str_value;
```

```
  }
| TYPE actual_parameters
  {
    type_spec.length = 4 + actual_parameters.length;
    actual_parameters.padding = type_spec.padding;
    actual_parameters.bcursor = type_spec.bcursor + 4 <= 80
      -> type_spec.bcursor + 4
      #  len(type_spec.padding) + 4 ;
    type_spec.ecursor = actual_parameters.ecursor;
    type_spec.str_value = type_spec.bcursor + 4 <= 80
      -> ["TYPE", actual_parameters.str_value]
      #  ["\n", type_spec.padding, "TYPE", actual_parameters.str_value];
  }
| FUNCTION actual_parameters
  {
    type_spec.length = 8 + actual_parameters.length;
    actual_parameters.bcursor = type_spec.bcursor + 8 <= 80
      -> type_spec.bcursor + 8
      #  len(type_spec.padding) + 8;
    actual_parameters.padding = type_spec.padding;
    type_spec.ecursor = actual_parameters.ecursor;
    type_spec.str_value = type_spec.bcursor + 8 <= 80
      -> ["FUNCTION", actual_parameters.str_value]
      #  ["\n", type_spec.padding, "FUNCTION", actual_parameters.str_value];
  }
| MACHINE actual_parameters
  {
    type_spec.length = 7 + actual_parameters.length;
    actual_parameters.bcursor = type_spec.bcursor + 7 <= 80
      -> type_spec.bcursor + 7
      #  len(type_spec.padding) + 7;
    actual_parameters.padding = type_spec.padding;
    actual_parameters.ecursor = actual_parameters.ecursor;
    type_spec.str_value = type_spec.bcursor + 7 <= 80
      -> ["MACHINE", actual_parameters.str_value]
      #  ["\n", type_spec.padding, "MACHINE", actual_parameters.str_value];
  }
| ITERATOR actual_parameters
  {
    type_spec.length = 8 + actual_parameters.length;
    actual_parameters.bcursor = type_spec.bcursor + 8 <= 80
      -> type_spec.bcursor + 8
      #  len(type_spec.padding) + 8;
    actual_parameters.padding = type_spec.padding;
    actual_parameters.ecursor = actual_parameters.ecursor;
    type_spec.str_value = type_spec.bcursor + 8 <= 80
      -> ["ITERATOR", actual_parameters.str_value]
      #  ["\n", type_spec.padding, "ITERATOR", actual_parameters.str_value];
  }
| '?'
  {
        type_spec.length = 1;
        type_spec.ecursor = type_spec.bcursor + 1 <= 80
          -> type_spec.bcursor + 1
          #  len(type_spec.padding) + 1;
        type_spec.str_value = type_spec.bcursor + 1 <= 80
          -> "?"
          #  ["\n", type_spec.padding, "?"];
      }
    ;

name_list
    : name_list NAME
      {
        name_list[1].length = name_list[2].length + len(NAME.%text);
        name_list[2].bcursor = name_list[1].bcursor;
        name_list[2].padding = name_list[1].padding;
        name_list[1].ecursor = (name_list[2].ecursor + len(NAME.%text) + 1) <= 80
          -> name_list[2].ecursor + len(NAME.%text) + 1
          # len(name_list[1].padding) + len(NAME.%text) + 1;
        name_list[1].str_value = (name_list[2].ecursor + len(NAME.%text) + 1) <= 80
          -> [name_list[2].str_value, NAME.%text, " "]
          #  [name_list[2].str_value, "\n", name_list[1].padding, NAME.%text, " "];
      }
    | NAME
      {
        name_list.length = len(NAME.%text);
        name_list.ecursor = (name_list.bcursor + len(NAME.%text) + 1) <= 80
          -> name_list.bcursor + len(NAME.%text) + 1
          #  len(name_list.padding) + len(NAME.%text) + 1;
        name_list.str_value = (name_list.bcursor + len(NAME.%text) + 1) <= 80
          -> [NAME.%text, spaces(1)]
          # ["\n", name_list.padding, NAME.%text, spaces(1)];
      }
    ;

optional_name
    : NAME formal_parameters
      {
        optional_name.length = optional_name.ecursor == formal_parameters.bcursor
          -> 0
          #  1;
        optional_name.ecursor = formal_parameters.ecursor;
        formal_parameters.indent = optional_name.indent;
        formal_parameters.bcursor = optional_name.bcursor + len(NAME.%text) < 80
          -> optional_name.bcursor + len(NAME.%text)
          #  len(optional_name.padding) + len(NAME.%text);
        formal_parameters.padding = [optional_name.padding, spaces(len(NAME.%text))];
        optional_name.str_value = optional_name.bcursor + len(NAME.%text) < 80
          -> [NAME.%text, formal_parameters.str_value]
          #  ["\n", optional_name.padding, NAME.%text, formal_parameters.str_value];
      }
    |
      {
        optional_name.length = 0;
        optional_name.ecursor = optional_name.bcursor;
        optional_name.str_value = "";
      }
    ;

parametrized_name
    : NAME actual_parameters
      {
        parametrized_name.length = len(NAME.%text) + actual_parameters.length;
        actual_parameters.bcursor = (parametrized_name.bcursor + len(NAME.%text)) <= 80
          -> parametrized_name.bcursor + len(NAME.%text)
          #  len(parametrized_name.padding) + len(NAME.%text);
        parametrized_name.ecursor = actual_parameters.ecursor;
        actual_parameters.padding = parametrized_name.padding;
        parametrized_name.str_value = (parametrized_name.bcursor +len(NAME.%text) ) <= 80
          -> [NAME.%text, actual_parameters.str_value]
          #  ["\n", parametrized_name.padding,  NAME.%text, actual_parameters.str_value];
      }
    ;

actual_parameters        | parameter values are determined at specification time
    : '{' arg_list '}'
      {
        actual_parameters.length = 2 + arg_list.length;
        arg_list.bcursor = actual_parameters.bcursor + 1 < 80
          -> actual_parameters.bcursor + 1
          #  len(actual_parameters.padding) + 1;
        arg_list.padding = spaces(arg_list.bcursor);
        actual_parameters.ecursor = arg_list.ecursor + 1;
        actual_parameters.str_value = actual_parameters.bcursor + 1 < 80
          -> ["{", arg_list.str_value, "}"]
          #  ["\n", actual_parameters.padding, "{", arg_list.str_value, "}"];
      }
    |       %prec SEMI               | must have a lower precedence than '{'
      {
        actual_parameters.length = 0;
        actual_parameters.ecursor = actual_parameters.bcursor;
        actual_parameters.str_value = "";
      }
    ;

actual_arguments              | arguments are evaluated at run-time
    : '(' arg_list ')'
      {
        actual_arguments.length = 2 + arg_list.length;
        arg_list.bcursor = actual_arguments.bcursor + 1 < 80
          -> actual_arguments.bcursor + 1
          #  len(actual_arguments.padding) + 1;
        arg_list.padding = spaces(arg_list.bcursor);
        actual_arguments.ecursor = arg_list.ecursor + 1;
        actual_arguments.str_value = actual_arguments.bcursor + 1 < 80
          -> ["(", arg_list.str_value, ")"]
          #  ["\n", actual_arguments.padding, "(", arg_list.str_value, ")"];
      }
    |       %prec SEMI           ! must have a lower precedence than '('
      {
        actual_arguments.length = 0;
        actual_arguments.ecursor = actual_arguments.bcursor;
        actual_arguments.str_value = "";
      }
    ;

arg_list
    : arg_list ',' arg              %prec COMMA
      {
        arg_list[1].length = arg_list[2].length + 2 + arg.length;
        arg_list[2].bcursor = arg_list[1].bcursor;
        arg_list[2].padding = arg_list[1].padding;
        arg.bcursor = arg_list[2].ecursor + 2 <= 80
          -> arg_list[2].ecursor + 2
          #  len(arg_list[1].padding);
        arg.padding = arg_list[1].padding;
        arg_list[1].ecursor = arg.ecursor;
        arg_list[1].str_value = arg_list[2].ecursor + 2 <= 80
          -> [arg_list[2].str_value, ", ", arg.str_value]
          #  [arg_list[2].str_value, ",", "\n", arg_list[1].padding, arg.str_value];
      }
    | arg
      {
        arg_list.length = arg.length;
        arg.bcursor = arg_list.bcursor;
        arg.padding = arg_list.padding;
        arg_list.ecursor = arg.ecursor;
        arg_list.str_value = arg.str_value;
      }
    ;

arg
    : expression
      {
        arg.length = expression.length;
        expression.bcursor = arg.bcursor;
        expression.padding = arg.padding;
        arg.ecursor = expression.ecursor;
        arg.str_value = expression.str_value;
      }
    | pair
      {
        arg.length = pair.length;
```

```
          pair.bcursor = arg.bcursor;
          pair.padding = arg.padding;
          arg.ecursor = pair.ecursor;
          arg.str_value = pair.str_value;
        }
      ;

expression_list
      : expression_list ',' comment expression            %prec COMMA
        {
          expression_list[1].length = expression_list[2].length + 2 + comment.length
            + expression.length;
          expression.bcursor = comment.length > 0
            -> len(expression_list[1].padding)
            #  expression_list[2].ecursor + 2 + expression.length <= 80
            -> expression_list[2].ecursor + 2
            #  len(expression_list[1].padding);
          comment.bcursor = expression_list[2].ecursor +1;
          expression.padding = expression_list[1].padding;
          expression_list[2].bcursor = expression_list[1].bcursor;
          expression_list[2].padding = expression_list[1].padding;
          expression_list[1].ecursor = expression.ecursor;
          expression_list[1].str_value = comment.length > 0
            -> [expression_list[2].str_value, ", ", comment.str_value,
                expression_list[1].padding, expression.str_value]
            #  expression_list[2].ecursor + 2 + expression.length <= 80
            -> [expression_list[2].str_value, ", ", comment.str_value, expression.str_value]
            #  [expression_list[2].str_value, ",", "\n", comment.str_value,
                expression_list[1].padding, expression.str_value];
        }
      | expression
        {
          expression_list.length = expression.length;
          expression.bcursor = expression_list.bcursor;
          expression.padding = expression_list.padding;
          expression_list.ecursor = expression.ecursor;
          expression_list.str_value = expression.str_value;
        }
      ;

expression
      : quantifier '(' field_list restriction BIND expression ')'
        {
          expression[1].length = quantifier.length + 2 + field_list.length
            + restriction.length + 4 + expression[2].length;
          quantifier.padding = expression[1].padding;
          quantifier.bcursor = expression[1].bcursor;
          field_list.bcursor = quantifier.ecursor + 1;
          field_list.padding = spaces(field_list.bcursor);
          restriction.bcursor = field_list.ecursor + restriction.length <= 80
            -> field_list.ecursor
      # field_list.bcursor;
    restriction.padding = field_list.padding;
    expression[2].bcursor = restriction.ecursor + 4 + expression[2].length + 1 <= 80
      -> restriction.ecursor + 4
      # len(restriction.padding) + 3;
    expression[2].padding = restriction.ecursor + 4 + expression[2].length + 1 <= 80
      -> field_list.padding
      # [restriction.padding,spaces(3)];
    expression[1].ecursor = expression[2].ecursor + 1;
    expression[1].str_value = (field_list.ecursor + restriction.length <= 80)
        && (restriction.ecursor + 4 + 1 + expression[2].length <= 80)
        && (restriction.length == 0)
      -> [quantifier.str_value, "(", field_list.str_value, restriction.str_value,
          " :: ", expression[2].str_value, ")"]
      # (field_list.ecursor + restriction.length <= 80)
        && (restriction.ecursor + 4 + 1 + expression[2].length <= 80)
        && (restriction.length > 0)
      -> [quantifier.str_value, "(", field_list.str_value, " ", restriction.str_value,
          " :: ", expression[2].str_value, ")"]
      # (field_list.ecursor + restriction.length > 80)
        && (restriction.ecursor + 4 + 1 + expression[2].length <= 80)
        && (restriction.length == 0)
      -> [quantifier.str_value, "(", field_list.str_value, "\n", field_list.padding,
          restriction.str_value, ":: ", expression[2].str_value, ")"]
      # (field_list.ecursor + restriction.length > 80)
        && (restriction.ecursor + 4 + 1 + expression[2].length <= 80)
        && (restriction.length > 0)
      -> [quantifier.str_value, "(", field_list.str_value, "\n", field_list.padding,
          restriction.str_value, " :: ", expression[2].str_value, ")"]
      # (field_list.ecursor + restriction.length > 80)
        && (restriction.ecursor + 4 + 1 + expression[2].length > 80)
        && (restriction.length > 0)
      -> [quantifier.str_value, "(", field_list.str_value, "\n", field_list.padding,
          restriction.str_value, "\n",restriction.padding,":: ",
          expression[2].str_value, ")"]
      # (field_list.ecursor + restriction.length > 80)
        && (restriction.ecursor + 4 + 1 + expression[2].length > 80)
        && (restriction.length == 0)
      -> [quantifier.str_value, "(", field_list.str_value, "\n",
          restriction.padding,":: ", expression[2].str_value, ")"]
      # restriction.length == 0
      -> [quantifier.str_value, "(", field_list.str_value, restriction.str_value,
         "\n", restriction.padding, ":: ", expression[2].str_value, ")"]
      # [quantifier.str_value, "(", field_list.str_value, " ", restriction.str_value,
         "\n", restriction.padding, ":: ", expression[2].str_value, ")"];
  }
| parametrized_name actual_arguments
  {
    expression.length = parametrized_name.length + actual_arguments.length;
    parametrized_name.bcursor = expression.bcursor;
    parametrized_name.padding = expression.padding;
```

```
      actual_arguments.bcursor = parametrized_name.ecursor;
      actual_arguments.padding = expression.padding;
      expression.ecursor = actual_arguments.ecursor;
      expression.str_value = [parametrized_name.str_value, actual_arguments.str_value];
    }
  | parametrized_name '@' parametrized_name actual_arguments
    {
      expression.length = parametrized_name[1].length + 1 + parametrized_name[2].length
        + actual_arguments.length;
      parametrized_name[1].bcursor = expression.bcursor;
      parametrized_name[1].padding = expression.padding;
      parametrized_name[2].bcursor = parametrized_name[1].ecursor + 1 <= 80
        -> parametrized_name[1].ecursor + 1
        #  len(parametrized_name[1].padding) + 1;
      parametrized_name[2].padding = expression.padding;
      actual_arguments.bcursor = parametrized_name[2].ecursor;
      actual_arguments.padding = expression.padding;
      expression.ecursor = actual_arguments.ecursor;
      expression.str_value = parametrized_name[1].ecursor + 1 <= 80
        -> [parametrized_name[1].str_value, "@", parametrized_name[2].str_value,
             actual_arguments.str_value]
        #  [parametrized_name[1].str_value,"\n", parametrized_name[1].padding, "@",
             parametrized_name[2].str_value, actual_arguments.str_value];
    }
  | NOT expression                  %prec NOT
    {
      expression[1].length = 1 + expression[2].length;
      expression[2].bcursor = expression[1].bcursor + 1 + expression[2].length <= 80
        -> expression[1].bcursor + 1
        #  len(expression[1].padding) +1;
      expression[2].padding = [expression[1].padding, " "];
      expression[1].ecursor = expression[2].ecursor;
      expression[1].str_value = expression[1].bcursor + 1 + expression[2].length <= 80
        -> ["~", expression[2].str_value]
        #  ["\n", expression[1].padding, "~", expression[2].str_value];
    }
  | expression AND expression      %prec AND
    {
      expression[1].length = expression[2].length + 3 + expression[3].length;
      expression[2].bcursor = expression[1].bcursor;
      expression[2].padding = expression[1].padding;
      expression[3].bcursor = expression[2].ecursor + 3 + expression[3].length <= 80
        -> expression[2].ecursor + 3
        #  len(expression[1].padding) + 2;
      expression[3].padding = expression[1].padding;
      expression[1].ecursor = expression[3].ecursor;
      expression[1].str_value = expression[2].ecursor + 3 + expression[3].length <= 80
        -> [expression[2].str_value, " & ", expression[3].str_value]
        #  [expression[2].str_value, "\n", expression[1].padding, "& ",
             expression[3].str_value];
    }
```

```
| expression OR expression        %prec OR
  {
    expression[1].length = expression[2].length + 3 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 3 + expression[3].length <= 80
      -> expression[2].ecursor + 3
      #  len(expression[1].padding) + 2;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 3 + expression[3].length <= 80
      -> [expression[2].str_value, " | ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "| ",
          expression[3].str_value];
  }
| expression IMPLIES expression        %prec IMPLIES
  {
    expression[1].length = expression[2].length + 4 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 4 + expression[3].length <= 80
      -> expression[2].ecursor + 4
      #  len(expression[1].padding) + 3;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 4 + expression[3].length <= 80
      -> [expression[2].str_value, " => ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "=> ",
          expression[3].str_value];
  }
| expression IFF expression      %prec IFF
  {
    expression[1].length = expression[2].length + 5 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 5 + expression[3].length <= 80
      -> expression[2].ecursor + 5
      #  len(expression[1].padding) + 4;
    expression[3].padding = spaces(expression[3].bcursor);
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 5 + expression[3].length <= 80
      -> [expression[2].str_value, " <=> ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "<=> ",
          expression[3].str_value];
  }
| expression '<' expression      %prec LE
  {
    expression[1].length = expression[2].length + 3 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 3 + expression[3].length <= 80
```

```
      -> expression[2].ecursor + 3
      #  len(expression[1].padding) + 2;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 3 + expression[3].length <= 80
      -> [expression[2].str_value, " < ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "< ",
          expression[3].str_value];
  }
| expression '>' expression      %prec LE
  {
    expression[1].length = expression[2].length + 3 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 3 + expression[3].length  <= 80
      -> expression[2].ecursor + 3
      #  len(expression[1].padding) + 2;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 3 + expression[3].length <= 80
      -> [expression[2].str_value, " > ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "> ",
          expression[3].str_value];
  }
| expression '=' expression      %prec LE
  {
    expression[1].length = expression[2].length + 3 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 3 + expression[3].length <= 80
      -> expression[2].ecursor + 3
      #  len(expression[1].padding) + 2;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 3 + expression[3].length <= 80
      -> [expression[2].str_value, " = ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "= ",
          expression[3].str_value];
  }
| expression LE expression      %prec LE
  {
    expression[1].length = expression[2].length + 4 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 4 + expression[3].length <= 80
      -> expression[2].ecursor + 4
      #  len(expression[1].padding) + 3;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 4 + expression[3].length <= 80
      -> [expression[2].str_value, " <= ", expression[3].str_value]
      # [expression[2].str_value, "\n", expression[1].padding, "<= ",
          expression[3].str_value];
  }
| expression GE expression       %prec LE
  {
    expression[1].length = expression[2].length + 4 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 4 + expression[3].length <= 80
      -> expression[2].ecursor + 4
      #  len(expression[1].padding) + 3;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 4 + expression[3].length <= 80
      -> [expression[2].str_value, " >= ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, ">= ",
          expression[3].str_value];
  }
| expression NE expression       %prec LE
  {
    expression[1].length = expression[2].length + 4 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 4 + expression[3].length <= 80
      -> expression[2].ecursor + 4
      #  len(expression[1].padding) + 3;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 4 + expression[3].length <= 80
      -> [expression[2].str_value, " ~= ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "~= ",
          expression[3].str_value];
  }
| expression NLT expression      %prec LE
  {
    expression[1].length = expression[2].length + 4 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 4 + expression[3].length <= 80
      -> expression[2].ecursor + 4
      #  len(expression[1].padding) + 3;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 4 + expression[3].length <= 80
      -> [expression[2].str_value, " ~< ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "~< ",
          expression[3].str_value];
  }
| expression NGT expression      %prec LE
  {
    expression[1].length = expression[2].length + 4 + expression[3].length;
```

```
      expression[2].bcursor = expression[1].bcursor;
      expression[2].padding = expression[1].padding;
      expression[3].bcursor = expression[2].ecursor + 4 + expression[3].length <= 80
        -> expression[2].ecursor + 4
        #  len(expression[1].padding) + 3;
      expression[3].padding = expression[1].padding;
      expression[1].ecursor = expression[3].ecursor;
      expression[1].str_value = expression[2].ecursor + 4 + expression[3].length <= 80
        -> [expression[2].str_value, " ~> ", expression[3].str_value]
        #  [expression[2].str_value, "\n", expression[1].padding, "~> ",
             expression[3].str_value];
    }
  | expression NLE expression      %prec LE
    {
      expression[1].length = expression[2].length + 5 + expression[3].length;
      expression[2].bcursor = expression[1].bcursor;
      expression[2].padding = expression[1].padding;
      expression[3].bcursor = expression[2].ecursor + 5 + expression[3].length <= 80
        -> expression[2].ecursor + 5
        #  len(expression[1].padding) + 4;
      expression[3].padding = expression[1].padding;
      expression[1].ecursor = expression[3].ecursor;
      expression[1].str_value = expression[2].ecursor + 5 + expression[3].length <= 80
        -> [expression[2].str_value, " ~<= ", expression[3].str_value]
        #  [expression[2].str_value, "\n", expression[1].padding, "~<= ",
             expression[3].str_value];
    }
  | expression NGE expression      %prec LE
    {
      expression[1].length = expression[2].length + 5 + expression[3].length;
      expression[2].bcursor = expression[1].bcursor;
      expression[2].padding = expression[1].padding;
      expression[3].bcursor = expression[2].ecursor + 5 + expression[3].length <= 80
        -> expression[2].ecursor + 5
        #  len(expression[1].padding) + 4;
      expression[3].padding = expression[1].padding;
      expression[1].ecursor = expression[3].ecursor;
      expression[1].str_value = expression[2].ecursor + 5 + expression[3].length <= 80
        -> [expression[2].str_value, " ~>= ", expression[3].str_value]
        #  [expression[2].str_value, "\n", expression[1].padding, "~>= ",
             expression[3].str_value];
    }
  | expression EQV expression      %prec LE
    {
      expression[1].length = expression[2].length + 4 + expression[3].length;
      expression[2].bcursor = expression[1].bcursor;
      expression[2].padding = expression[1].padding;
      expression[3].bcursor = expression[2].ecursor + 4 + expression[3].length <= 80
        -> expression[2].ecursor + 4
        #  len(expression[1].padding) + 3;
      expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 4 + expression[3].length <= 80
      -> [expression[2].str_value, " == ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "== ",
          expression[3].str_value];
  }
| expression NEQV expression     %prec LE
  {
    expression[1].length = expression[2].length + 5 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 5 + expression[3].length <= 80
      -> expression[2].ecursor + 5
      #  len(expression[1].padding) + 4;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 5 + expression[3].length <= 80
      -> [expression[2].str_value, " ~== ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "~== ",
          expression[3].str_value];
  }
| '-' expression                 %prec UMINUS
  {
    expression[1].length = 1 + expression[2].length;
    expression[2].bcursor = expression[1].bcursor + 1 + expression[2].length <= 80
      -> expression[1].bcursor + 1
      #  len(expression[1].padding) + 1;
    expression[2].padding = expression[1].padding;
    expression[1].ecursor = expression[2].ecursor;
    expression[1].str_value = expression[1].bcursor + 1 + expression[2].length <= 80
      -> ["-", expression[2].str_value]
      #   ["\n", expression[1].padding, "-", expression[2].str_value];
  }
| expression '+' expression      %prec PLUS
  {
    expression[1].length = expression[2].length + 3 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 3 + expression[3].length <= 80
      -> expression[2].ecursor + 3
      #  len(expression[1].padding) + 2;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 3 + expression[3].length <= 80
      -> [expression[2].str_value, " + ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "+ ",
          expression[3].str_value];
  }
| expression '-' expression      %prec MINUS
  {
    expression[1].length = expression[2].length + 3 + expression[3].length;
```

```
      expression[2].bcursor = expression[1].bcursor;
      expression[2].padding = expression[1].padding;
      expression[3].bcursor = expression[2].ecursor + 3 + expression[3].length <= 80
        -> expression[2].ecursor + 3
        #  len(expression[1].padding) + 2;
      expression[3].padding = expression[1].padding;
      expression[1].ecursor = expression[3].ecursor;
      expression[1].str_value = expression[2].ecursor + 3 + expression[3].length <= 80
        -> [expression[2].str_value, " - ", expression[3].str_value]
        #  [expression[2].str_value, "\n", expression[1].padding, "- ",
             expression[3].str_value];
    }
  | expression '*' expression       %prec MUL
    {
      expression[1].length = expression[2].length + 3 + expression[3].length;
      expression[2].bcursor = expression[1].bcursor;
      expression[2].padding = expression[1].padding;
      expression[3].bcursor = expression[2].ecursor + 3 + expression[3].length <= 80
        -> expression[2].ecursor + 3
        #  len(expression[1].padding) + 2;
      expression[3].padding = expression[1].padding;
      expression[1].ecursor = expression[3].ecursor;
      expression[1].str_value = expression[2].ecursor + 3 + expression[3].length <= 80
        -> [expression[2].str_value, " * ", expression[3].str_value]
        #  [expression[2].str_value, "\n", expression[1].padding, "* ",
             expression[3].str_value];
    }
  | expression '/' expression       %prec DIV
    {
      expression[1].length = expression[2].length + 3 + expression[3].length;
      expression[2].bcursor = expression[1].bcursor;
      expression[2].padding = expression[1].padding;
      expression[3].bcursor = expression[2].ecursor + 3 + expression[3].length <= 80
        -> expression[2].ecursor + 3
        #  len(expression[1].padding) + 2;
      expression[3].padding = expression[1].padding;
      expression[1].ecursor = expression[3].ecursor;
      expression[1].str_value = expression[2].ecursor + 3 + expression[3].length <= 80
        -> [expression[2].str_value, " / ", expression[3].str_value]
        #  [expression[2].str_value, "\n", expression[1].padding, "/ ",
             expression[3].str_value];
    }
  | expression MOD expression       %prec MOD
    {
      expression[1].length = expression[2].length + 5 + expression[3].length;
      expression[2].bcursor = expression[1].bcursor;
      expression[2].padding = expression[1].padding;
      expression[3].bcursor = expression[2].ecursor + 5 + expression[3].length <= 80
        -> expression[2].ecursor + 5
        #  len(expression[1].padding) + 4;
      expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 5 + expression[3].length <= 80
      -> [expression[2].str_value, " MOD ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "MOD ",
           expression[3].str_value];
  }
| expression EXP expression      %prec EXP
  {
    expression[1].length = expression[2].length + 4 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 4 + expression[3].length <= 80
      -> expression[2].ecursor + 4
      #  len(expression[1].padding) + 3;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 4 + expression[3].length <= 80
      -> [expression[2].str_value, " ** ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "** ",
           expression[3].str_value];
  }
| expression U expression        %prec U
  {
    expression[1].length = expression[2].length + 3 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 3 + expression[3].length <= 80
      -> expression[2].ecursor + 3
      #  len(expression[1].padding) + 2;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 3 + expression[3].length <= 80
      -> [expression[2].str_value, " U ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "U ",
           expression[3].str_value];
  }
| expression APPEND expression          %prec APPEND
  {
    expression[1].length = expression[2].length + 4 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 4 + expression[3].length <= 80
      -> expression[2].ecursor + 4
      #  len(expression[1].padding) + 3;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 4 + expression[3].length <= 80
      -> [expression[2].str_value, " || ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "|| ",
           expression[3].str_value];
  }
```

```
¦ expression IN expression      %prec IN
  {
    expression[1].length = expression[2].length + 4 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 4 + expression[3].length <= 80
      -> expression[2].ecursor + 4
      #  len(expression[1].padding) + 3;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 4 + expression[3].length <= 80
      -> [expression[2].str_value, " IN ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, "IN ",
           expression[3].str_value];
  }
¦ '*' expression                %prec STAR
  | *x is the value of x before a transition
  | x is the value after the transition
  {
    expression[1].length = 1 + expression[2].length;
    expression[2].bcursor = expression[1].bcursor + 1 + expression[2].length <= 80
      -> expression[1].bcursor + 1
      #  len(expression[1].padding) + 1;
    expression[2].padding = expression[1].padding;
    expression[1].ecursor = expression[2].ecursor;
    expression[1].str_value = expression[1].bcursor + 1 + expression[2].length <= 80
      -> ["*", expression[2].str_value]
      #  ["\n", expression[1].padding, "*", expression[2].str_value];
  }
¦ '$' expression                %prec DOT
  | $x represents a collection of items rather than just one
  | s1 = {x, $s2} means s1 = union({x}, s2)
  | s1 = [x, $s2] means s1 = append([x], s2)
  {
    expression[1].length = 1 + expression[2].length;
    expression[2].bcursor = expression[1].bcursor + 1 + expression[2].length <= 80
      -> expression[1].bcursor + 1
      #  len(expression[1].padding) + 1;
    expression[2].padding = expression[1].padding;
    expression[1].ecursor = expression[2].ecursor;
    expression[1].str_value = expression[1].bcursor + 1 + expression[2].length <= 80
      -> ["$", expression[2].str_value]
      #  ["\n", expression[1].padding, "$", expression[2].str_value];
  }
¦ expression RANGE expression          %prec RANGE
  | x in [a .. b] iff x in {a .. b} iff a <= x <= b
  | [a .. b] is sorted in increasing order
  {
    expression[1].length = expression[2].length + 4 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 4 + expression[3].length <= 80
      -> expression[2].ecursor + 4
      #  len(expression[1].padding) + 3;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor;
    expression[1].str_value = expression[2].ecursor + 4 + expression[3].length <= 80
      -> [expression[2].str_value, " .. ", expression[3].str_value]
      #  [expression[2].str_value, "\n", expression[1].padding, ".. ",
          expression[3].str_value];
  }
| expression '.' NAME            %prec DOT
  {
    expression[1].length = expression[2].length + 1 + len(NAME.%text);
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[1].ecursor = expression[2].ecursor + 1 + len(NAME.%text) <= 80
      -> expression[2].ecursor + 1 + len(NAME.%text)
      #  len(expression[1].padding) + len(NAME.%text);
    expression[1].str_value = expression[2].ecursor + 1 + len(NAME.%text) <= 80
      -> [expression[2].str_value, ".", NAME.%text]
      #  [expression[2].str_value, ".", "\n", expression[1].padding, NAME.%text];
  }
| expression '[' expression ']'         %prec DOT
  {
    expression[1].length = expression[2].length + 2 + expression[3].length;
    expression[2].bcursor = expression[1].bcursor;
    expression[2].padding = expression[1].padding;
    expression[3].bcursor = expression[2].ecursor + 2 + expression[3].length <= 80
      -> expression[2].ecursor + 1
      #  len(expression[1].padding) + 1;
    expression[3].padding = expression[1].padding;
    expression[1].ecursor = expression[3].ecursor + 1;
    expression[1].str_value = expression[2].ecursor + 2 + expression[3].length <= 80
      -> [expression[2].str_value, "[", expression[3].str_value, "]"]
      #  [expression[2].str_value, "\n", expression[1].padding, "[",
          expression[3].str_value, "]"];
  }
| '(' expression ')'
  {
    expression[1].length = 2 + expression[2].length;
    expression[2].bcursor = expression[1].bcursor + 2 + expression[2].length <= 80
      -> expression[1].bcursor + 1
      #  len(expression[1].padding) + 1;
    expression[2].padding = expression[1].padding;
    expression[1].ecursor = expression[2].ecursor + 1;
    expression[1].str_value = expression[1].bcursor + 2 + expression[2].length <= 80
      -> ["(", expression[2].str_value, ")"]
      #  ["\n", expression[1].padding, "(", expression[2].str_value, ")"];
  }
| '(' expression units ')'       ! timing expression
  {
      expression[1].length = 2 + expression[2].length + units.length;
      expression[2].bcursor = expression[1].bcursor + 2 + expression[2].length
        + units.length <= 80
        -> expression[1].bcursor + 1
        #  len(expression[1].padding) + 1;
      expression[2].padding = expression[1].padding;
      units.bcursor = expression[2].ecursor;
      units.padding = expression[1].padding;
      expression[1].ecursor = units.ecursor + 1;
      expression[1].str_value = expression[1].bcursor + 2 + expression[2].length
        + units.length <= 80
        -> ["(", expression[2].str_value, units.str_value, ")"]
        #  ["\n", expression[1].padding, "(", expression[2].str_value, units.str_value,
            ")"];
    }
  | TIME                  | The current local time, used in temporal events
    {
      expression.length = 5;
      expression.ecursor = expression.bcursor + 5 <= 80
        -> expression.bcursor + 5
        #  len(expression.padding) + 5;
      expression.str_value = expression.bcursor + 5 <= 80
        -> "TIME "
        #  ["\n", expression.padding, "TIME "];
    }
  | DELAY                 | The time between the triggering event and the response
    {
      expression.length = 6;
      expression.ecursor = expression.bcursor + 6 <= 80
        -> expression.bcursor + 6
        #  len(expression.padding) + 6;
      expression.str_value = expression.bcursor + 6 <= 80
        -> "DELAY "
        #  ["\n", expression.padding, "DELAY "];
    }
  | PERIOD                | The time between successive events of this type
    {
      expression.length = 7;
      expression.ecursor = expression.bcursor + 7 <= 80
        -> expression.bcursor + 7
        #  len(expression.padding) + 7;
      expression.str_value = expression.bcursor + 7 <= 80
        -> "PERIOD "
        #  ["\n", expression.padding, "PERIOD "];
    }
  | literal
    {
      expression.length = literal.length;
      literal.bcursor = expression.bcursor;
      literal.padding = expression.padding;
      expression.ecursor = literal.ecursor;
      expression.str_value = literal.str_value;
    }
  | literal '@' parametrized_name          ! literal with explicit type
    {
      expression.length = literal.length + 1 + parametrized_name.length;
      literal.bcursor = expression.bcursor;
      literal.padding = expression.padding;
      parametrized_name.bcursor = literal.ecursor + 1 + parametrized_name.length <= 80
        -> literal.ecursor + 1
        #  len(literal.padding);
      parametrized_name.padding = spaces(parametrized_name.bcursor);
      expression.ecursor = parametrized_name.ecursor;
      expression.str_value = literal.ecursor + 1 + parametrized_name.length <= 80
        -> [literal.str_value, "@", parametrized_name.str_value]
        #  [literal.str_value, "@", "\n", literal.padding, parametrized_name.str_value];
    }
  | '?'                    ! An undefined value to be specified later
    {
      expression.length = 1;
      expression.ecursor = expression.bcursor + 1 <= 80
        -> expression.bcursor + 1
        #  len(expression.padding) + 1;
      expression.str_value = expression.bcursor + 1 <= 80
        -> "?"
        #  ["\n", expression.padding, "?"];
    }
  | '!'                    ! An undefined and illegal value
    {
      expression.length = 1;
      expression.ecursor = expression.bcursor + 1 <= 80
        -> expression.bcursor + 1
        #  len(expression.padding) + 1;
      expression.str_value = expression.bcursor + 1 <= 80
        -> "!"
        #  ["\n", expression.padding, "!"];
    }
  | IF expression THEN expression middle_cases ELSE expression FI
    {
      expression[1].length = 15 + expression[2].length + expression[3].length
        + middle_cases.length + expression[4].length;
      expression[2].bcursor = expression[1].bcursor + 3;
      expression[2].padding = spaces(expression[1].bcursor);
      expression[3].bcursor = expression[1].bcursor + 5;
      expression[3].padding = spaces(expression[1].bcursor);
      middle_cases.bcursor = expression[1].bcursor;
      middle_cases.padding = spaces(expression[1].bcursor);
      expression[4].bcursor = expression[1].bcursor + 5;
      expression[4].padding = spaces(expression[1].bcursor);
      expression[1].ecursor = expression[4].ecursor + 4;
      expression[1].str_value = expression[4].ecursor + 4 <= 80
        -> ["IF ", expression[2].str_value, "\n", expression[3].padding, "THEN ",
              expression[3].str_value, middle_cases.str_value, "\n",
              expression[4].padding, "ELSE ", expression[4].str_value, " FI "]
          #  ["IF ", expression[2].str_value, "\n", expression[3].padding, "THEN ",
              expression[3].str_value, middle_cases.str_value, "\n",
              expression[4].padding, "ELSE ", expression[4].str_value, "\n",
              expression[4].padding, " FI "];
        }
      ;

middle_cases
      : middle_cases ELSE_IF expression THEN expression
        {
          middle_cases[1].length = 13 + middle_cases[2].length + expression[1].length
            + expression[2].length;
          middle_cases[2].bcursor = middle_cases[1].bcursor;
          middle_cases[2].padding = middle_cases[1].padding;
          expression[1].bcursor = middle_cases[1].bcursor + 8;
          expression[1].padding = middle_cases[1].padding;
          expression[2].bcursor = middle_cases[1].bcursor + 5;
          expression[2].padding = middle_cases[1].padding;
          middle_cases[1].str_value = [middle_cases[2].str_value, "\n",
            middle_cases[1].padding, "ELSE_IF ", expression[1].str_value, "\n",
            middle_cases[1].padding, "THEN ", expression[2].str_value];
        }
      |
        {
          middle_cases.length = 0;
          middle_cases.str_value = "";
        }
      ;

quantifier
      : ALL
        {
          quantifier.length = 3;
          quantifier.ecursor = quantifier.bcursor <= 80
            -> quantifier.bcursor + 3
            #  len(quantifier.padding) + 3;
          quantifier.str_value = quantifier.bcursor <= 80
            -> "ALL"
            #   ["\n", quantifier.padding, "ALL"];
        }
      | SOME
        {
          quantifier.length = 4;
          quantifier.ecursor = quantifier.bcursor <= 80
            -> quantifier.bcursor + 4
            #  len(quantifier.padding) + 4;
          quantifier.str_value = quantifier.bcursor <= 80
            -> "SOME"
            #  ["\n", quantifier.padding, "SOME"];
```

```
  }
| NUMBER
  {
    quantifier.length = 6;
    quantifier.ecursor = quantifier.bcursor <= 80
      -> quantifier.bcursor + 6
      #  len(quantifier.padding) + 6;
    quantifier.str_value = quantifier.bcursor <= 80
      -> "NUMBER"
      #  ["\n", quantifier.padding, "NUMBER"];
  }
| SUM
  {
    quantifier.length = 3;
    quantifier.ecursor = quantifier.bcursor <= 80
      -> quantifier.bcursor + 3
      #  len(quantifier.padding) + 3;
    quantifier.str_value = quantifier.bcursor <= 80
      -> "SUM"
      #  ["\n", quantifier.padding, "SUM"];
  }
| PRODUCT
  {
    quantifier.length = 7;
    quantifier.ecursor = quantifier.bcursor <= 80
      -> quantifier.bcursor + 7
      #  len(quantifier.padding) + 7;
    quantifier.str_value = quantifier.bcursor <= 80
      -> "PRODUCT"
      #  ["\n", quantifier.padding, "PRODUCT"];
  }
| SET
  {
    quantifier.length = 3;
    quantifier.ecursor = quantifier.bcursor <= 80
      -> quantifier.bcursor + 3
      #  len(quantifier.padding) + 3;
    quantifier.str_value = quantifier.bcursor <= 80
      -> "SET"
      #  ["\n", quantifier.padding, "SET"];
  }
| MAXIMUM
  {
    quantifier.length = 7;
    quantifier.ecursor = quantifier.bcursor <= 80
      -> quantifier.bcursor + 7
      #  len(quantifier.padding) + 7;
    quantifier.str_value = quantifier.bcursor <= 80
      -> "MAXIMUM"
      #  ["\n", quantifier.padding, "MAXIMUM"];
  }
```

```
    | MINIMUM
      {
        quantifier.length = 7;
        quantifier.ecursor = quantifier.bcursor <= 80
          -> quantifier.bcursor + 7
          #  len(quantifier.padding) + 7;
        quantifier.str_value = quantifier.bcursor <= 80
          -> "MINIMUM"
          #  ["\n", quantifier.padding, "MINIMUM"];
      }
    | UNION
      {
        quantifier.length = 5;
        quantifier.ecursor = quantifier.bcursor <= 80
          -> quantifier.bcursor + 5
          #  len(quantifier.padding) + 5;
        quantifier.str_value = quantifier.bcursor <= 80
          -> "UNION"
          #  ["\n", quantifier.padding, "UNION"];
      }
    | INTERSECTION
      {
        quantifier.length = 12;
        quantifier.ecursor = quantifier.bcursor <= 80
          -> quantifier.bcursor + 12
          #  len(quantifier.padding) + 12;
        quantifier.str_value = quantifier.bcursor <= 80
          -> "INTERSECTION"
          #  ["\n", quantifier.padding, "INTERSECTION"];
      }
    ;

restriction
    : SUCH expression
      {
        restriction.length = 10 + expression.length;
        expression.bcursor = restriction.bcursor + 10 <= 80
          -> restriction.bcursor + 10
          #  len(restriction.padding) + 10;
        expression.padding = spaces(expression.bcursor);
        restriction.ecursor = expression.ecursor;
        restriction.str_value = restriction.bcursor + 10 <= 80
          -> ["SUCH THAT ", expression.str_value]
          #  ["\n", restriction.padding, "SUCH THAT ", expression.str_value];
      }
    |
      {
        restriction.length = 0;
        restriction.ecursor = restriction.bcursor;
        restriction.str_value = "";
      }
    ;

literal
    : INTEGER_LITERAL
      {
        literal.length = len(INTEGER_LITERAL.%text);
        literal.str_value = literal.bcursor + len(INTEGER_LITERAL.%text) <= 80
          -> INTEGER_LITERAL.%text
          #  ["\n", literal.padding, INTEGER_LITERAL.%text];
        literal.ecursor = literal.bcursor + len(INTEGER_LITERAL.%text) <= 80
          -> literal.bcursor + len(INTEGER_LITERAL.%text)
          #  len( literal.padding) + len(INTEGER_LITERAL.%text);
      }
    | REAL_LITERAL
      {
        literal.length = len(REAL_LITERAL.%text);
        literal.str_value = literal.bcursor + len(REAL_LITERAL.%text) <= 80
          -> REAL_LITERAL.%text
          #  ["\n", literal.padding, REAL_LITERAL.%text];
        literal.ecursor = literal.bcursor + len(REAL_LITERAL.%text) <= 80
          -> literal.bcursor + len(REAL_LITERAL.%text)
          #  len(literal.padding) + len(REAL_LITERAL.%text);
      }
    | CHAR_LITERAL
      {
        literal.length = len(CHAR_LITERAL.%text);
        literal.str_value = literal.bcursor + len(CHAR_LITERAL.%text) <= 80
          -> CHAR_LITERAL.%text
          #  ["\n", literal.padding, CHAR_LITERAL.%text];
        literal.ecursor = literal.bcursor + len(CHAR_LITERAL.%text) <= 80
          -> literal.bcursor + len(CHAR_LITERAL.%text)
          #  len(literal.padding) + len(CHAR_LITERAL.%text);
      }
    | STRING_LITERAL
      {
        literal.length = len(STRING_LITERAL.%text);
        literal.str_value = literal.bcursor + len(STRING_LITERAL.%text) <= 80
          -> STRING_LITERAL.%text
          #  ["\n", literal.padding, STRING_LITERAL.%text];
        literal.ecursor = literal.bcursor + len(STRING_LITERAL.%text) <= 80
          -> literal.bcursor + len(STRING_LITERAL.%text)
          #  len(literal.padding) + len(STRING_LITERAL.%text);
      }
    | '#' NAME               ! enumeration type literal
      {
        literal.length = 1 + len(NAME.%text);
        literal.ecursor = literal.bcursor + 1 + len(NAME.%text) <= 80
          -> literal.bcursor + 1 + len(NAME.%text)
          #  len(literal.padding) + 1 + len(NAME.%text);
        literal.str_value = literal.bcursor + 1 +len(NAME.%text) <= 80
          -> ["#", NAME.%text]
        # ["\n", literal.padding, "#", NAME.%text];
    }
  | '[' expressions ']'              | sequence literal
    {
      literal.length = 2 + expressions.length;
      expressions.bcursor = literal.bcursor + 1  < 80
        -> literal.bcursor + 1
        #  len(literal.padding) + 1;
      expressions.padding = spaces(literal.bcursor + 1);
      literal.ecursor = expressions.ecursor + 1;
      literal.str_value = literal.bcursor + 1  < 80
        -> ["[", expressions.str_value, "]"]
        #  ["\n", literal.padding, "[", expressions.str_value, "]"];
    }
  | '{' expressions '}'              | set literal
    {
      literal.length = 2 + expressions.length;
      expressions.bcursor = literal.bcursor + 1  < 80
        -> literal.bcursor + 1
        #  len(literal.padding) + 1;
      expressions.padding = spaces(literal.bcursor + 1);
      literal.ecursor = expressions.ecursor + 1;
      literal.str_value = literal.bcursor + 1  < 80
        -> ["{", expressions.str_value, "}"]
        #  ["\n", literal.padding, "{", expressions.str_value, "}"];
    }
  | '{' expression ';' comment expressions '}'          | map literal
    {
      literal.length = 3 + expression.length + comment.length + expressions.length;
      comment.bcursor = expression.ecursor + 1;
      expression.bcursor = literal.bcursor + 1  < 80
        -> literal.bcursor + 1
        #  len(literal.padding) + 1;
      expression.padding = spaces(expression.bcursor);
      expressions.bcursor = comment.length > 0
        -> len(expression.padding)
        #  expression.ecursor + 2  < 80
        -> expression.ecursor + 1
        #  len(expression.padding);
      expressions.padding = expression.padding;
      literal.ecursor = expressions.ecursor + 1;
      literal.str_value = (comment.length > 0) && (literal.bcursor + 1  < 80)
        -> ["{", expression.str_value, ";", comment.str_value, expression.padding,
             expressions.str_value, "}"]
        #  (comment.length > 0) && (literal.bcursor + 1  >= 80)
        -> ["\n", literal.padding, "{", expression.str_value, ";", comment.str_value,
             expression.padding, expressions.str_value, "}"]
        #  (comment.length == 0) && (literal.bcursor + 1  < 80)
          && (expression.ecursor + 3  <= 80)
        -> ["{", expression.str_value, "; ", comment.str_value, expressions.str_value,
             "}"]
          *  (comment.length == 0) && (literal.bcursor + 1  >= 80)
            &&  (expression.ecursor + 3  <= 80)
          -> ["\n", literal.padding, "{", expression.str_value, "; ", comment.str_value,
              expressions.str_value, "}"]
          *  (comment.length == 0) && (literal.bcursor + 1 < 80)
            &&  (expression.ecursor + 3  <= 80)
          -> ["{", expression.str_value, "; ", "\n", expression.padding,
              comment.str_value, expressions.str_value, "}"]
          *  ["\n", literal.padding, "{", expression.str_value, "; ", "\n",
              expression.padding, comment.str_value, expressions.str_value, "}"];
      }
    | '[' pair_list ']'               | tuple literal
      {
        literal.length = 2 + pair_list.length;
        pair_list.bcursor = literal.bcursor + 1  < 80
          -> literal.bcursor + 1
          *  len(literal.padding) + 1;
        pair_list.padding = spaces(literal.bcursor + 1);
        literal.ecursor = pair_list.ecursor + 1;
        literal.str_value = literal.bcursor + 1  < 80
          ->  ["[", pair_list.str_value, "]"]
          *   ["\n", literal.padding, "[", pair_list.str_value, "]"];
      }
    | '{' pair '}'                    | one_of literal
      {
        literal.length = 2 + pair.length;
        pair.bcursor = literal.bcursor + 1 < 80
          -> literal.bcursor + 1
          *  len(literal.padding) + 1;
        pair.padding = spaces(literal.bcursor + 1);
        literal.ecursor = pair.ecursor + 1;
        literal.str_value = literal.bcursor + 1  < 80
          -> ["{", pair.str_value, "}"]
          *  ["\n", literal.padding, "{", pair.str_value, "}"];
      }
    ;
      | relation literals are sets of tuples

expressions
    : expression_list
      {
        expressions.length = expression_list.length;
        expression_list.bcursor = expressions.bcursor;
        expression_list.padding = expressions.padding;
        expressions.ecursor = expression_list.ecursor;
        expressions.str_value = expression_list.str_value;
      }
    |
      {
        expressions.length = 0;
        expressions.ecursor = expressions.bcursor;
        expressions.str_value = "";
      }
    ;

pair_list
    : pair_list ',' pair
      {
        pair_list[1].length = pair_list[2].length + 2 + pair.length;
        pair_list[2].bcursor = pair_list[1].bcursor;
        pair_list[1].ecursor = pair.ecursor;
        pair_list[2].padding = pair_list[1].padding;
        pair.bcursor = pair_list[2].ecursor + 2 <= 80
          -> pair_list[2].ecursor + 2
          #  len(pair_list[1].padding);
        pair_list[1].str_value = pair_list[2].ecursor + 2 <= 80
          -> [pair_list[2].str_value, ", ", pair.str_value]
          #  [pair_list[2].str_value, ", ", "\n", pair_list[1].padding, pair.str_value];
        pair.padding = pair_list[1].padding;
      }
    | NAME pair
      {
        pair_list.length = len(NAME.%text) + pair.length;
        pair.bcursor = pair_list.bcursor + len(NAME.%text) <= 80
          -> pair_list.bcursor + len(NAME.%text)
          #  len(pair_list.padding) + len(NAME.%text);
        pair.padding = pair_list.padding;
        pair_list.ecursor = pair.ecursor;
        pair_list.str_value = pair_list.bcursor + len(NAME.%text) <= 80
          -> [NAME.%text, pair.str_value]
          #  ["\n", pair_list.padding, NAME.%text, pair.str_value];
      }
    | pair
      {
        pair_list.length = pair.length;
        pair.bcursor = pair_list.bcursor;
        pair.padding = pair_list.padding;
        pair_list.ecursor = pair.ecursor;
        pair_list.str_value = pair.str_value;
      }
    ;

pair
    : NAME BIND expression
      {
        pair.length = len(NAME.%text) + 4 + expression.length;
        expression.bcursor = pair.bcursor + len(NAME.%text) + 4 <= 80
          -> pair.bcursor + len(NAME.%text) + 4
          #  len(pair.padding) + len(NAME.%text) + 4;
        pair.ecursor = expression.ecursor;
        expression.padding = pair.padding;
        pair.str_value = pair.bcursor + len(NAME.%text) + 4 <= 80
        -> [NAME.%text, " :: ", expression.str_value]
        #  ["\n", pair.padding, NAME.%text, " :: ", expression.str_value];
      }
    ;

units
    : NANOSEC
      {
        units.length = 8;
        units.ecursor = units.bcursor + 8 < 80
          -> units.bcursor + 8
          #  len(units.padding) + 8;
        units.str_value = units.bcursor + 8 < 80
          -> "NANOSEC"
          #  ["\n", units.padding, "NANOSEC"];
      }
    | MICROSEC
      {
        units.length = 8;
        units.ecursor = units.bcursor + 8 < 80
          -> units.bcursor + 8
          #  len(units.padding) + 8;
        units.str_value = units.bcursor + 8 < 80
          -> "MICROSEC"
          #  ["\n", units.padding, "MICROSEC"];
      }
    | MILLISEC
      {
        units.length = 8;
        units.ecursor = units.bcursor + 8 < 80
          -> units.bcursor + 8
          #  len(units.padding) + 8;
        units.str_value = units.bcursor + 8 < 80
          -> "MILLISEC"
          #  ["\n", units.padding, "MILLISEC"];
      }
    | SECONDS
      {
        units.length = 7;
        units.ecursor = units.bcursor + 7 < 80
          -> units.bcursor + 7
          #  len(units.padding) + 7;
        units.str_value = units.bcursor + 7 < 80
          -> "SECONDS"
          #  ["\n", units.padding, "SECONDS"];
      }
    | MINUTES
      {
        units.length = 7;
        units.ecursor = units.bcursor + 7 < 80
          -> units.bcursor + 7
          #  len(units.padding) + 7;
        units.str_value = units.bcursor + 7 < 80
          -> "MINUTES"
          #  ["\n", units.padding, "MINUTES"];
      }
    | HOURS
      {
        units.length = 5;
        units.ecursor = units.bcursor + 5 < 80
          -> units.bcursor + 5
          #  len(units.padding) + 5;
        units.str_value = units.bcursor + 5 < 80
          -> "HOURS"
          #  ["\n", units.padding, "HOURS"];
      }
    | DAYS
      {
        units.length = 4;
        units.ecursor = units.bcursor + 4 < 80
          -> units.bcursor + 4
          #  len(units.padding) + 4;
        units.str_value = units.bcursor + 4 < 80
          -> "DAYS"
          #  ["\n", units.padding, "DAYS"];
      }
    | WEEKS
      {
        units.length = 5;
        units.ecursor = units.bcursor + 5 < 80
          -> units.bcursor + 5
          #  len(units.padding) + 5;
        units.str_value = units.bcursor + 5 < 80
          -> "WEEKS"
          #  ["\n", units.padding, "WEEKS"];
      }
    ;

operator_list
    : operator_list operator_symbol
      {
        operator_list[1].length = operator_list[2].length + operator_symbol.length;
        operator_list[2].bcursor = operator_list[1].bcursor;
        operator_list[2].padding = operator_list[1].padding;
        operator_list[1].ecursor = operator_symbol.ecursor;
        operator_symbol.padding = operator_list[1].padding;
        operator_symbol.bcursor = operator_list[2].ecursor;
        operator_list[1].str_value = [operator_list[2].str_value,
          operator_symbol.str_value ];
      }
    | operator_symbol
      {
        operator_list.length = operator_symbol.length;
        operator_list.ecursor = operator_symbol.ecursor;
        operator_symbol.bcursor = operator_list.bcursor;
        operator_symbol.padding = operator_list.padding;
        operator_list.str_value = operator_symbol.str_value;
      }
    ;

operator_symbol
    : NOT
      {
        operator_symbol.length = 2;
        operator_symbol.str_value = operator_symbol.bcursor + 2 <= 80
          -> "~ "
          #  ["\n", operator_symbol.padding, "~ "];
        operator_symbol.ecursor = operator_symbol.bcursor + 2 <= 80
          ->  operator_symbol.bcursor + 2
          #   len(operator_symbol.padding) + 2;
      }
    | AND
      {
        operator_symbol.length = 2;
        operator_symbol.str_value = operator_symbol.bcursor + 2 <= 80
          -> "& "
          #  ["\n", operator_symbol.padding, "& "];
        operator_symbol.ecursor = operator_symbol.bcursor + 2 <= 80
          -> operator_symbol.bcursor + 2
          #  len(operator_symbol.padding) + 2;
      }
    | OR
      {
        operator_symbol.length = 2;
        operator_symbol.str_value = operator_symbol.bcursor + 2 <= 80
          -> "| "
          #  ["\n", operator_symbol.padding, "| "];
        operator_symbol.ecursor = operator_symbol.bcursor + 2 <= 80
          -> operator_symbol.bcursor + 2
          #  len(operator_symbol.padding) + 2;
      }
    | IMPLIES
      {
        operator_symbol.length = 3;
        operator_symbol.str_value = operator_symbol.bcursor + 3 <= 80
          -> "=> "
          #  ["\n", operator_symbol.padding, "=> "];
        operator_symbol.ecursor = operator_symbol.bcursor + 3 <= 80
          -> operator_symbol.bcursor + 3
          #  len(operator_symbol.padding) + 3;
      }
    | IFF
      {
```

```
    operator_symbol.length = 4;
    operator_symbol.str_value = operator_symbol.bcursor + 4 <= 80
      -> "<=> "
      #  ["\n", operator_symbol.padding, "<=> "];
    operator_symbol.ecursor = operator_symbol.bcursor + 4 <= 80
      -> operator_symbol.bcursor + 4
      #  len(operator_symbol.padding) + 4;
  }
| '<'
  {
    operator_symbol.length = 2;
    operator_symbol.str_value = operator_symbol.bcursor + 2 <= 80
      -> "< "
      #  ["\n", operator_symbol.padding, "< "];
    operator_symbol.ecursor = operator_symbol.bcursor + 2 <= 80
      -> operator_symbol.bcursor + 2
      #  len(operator_symbol.padding) + 2;
  }
| '>'
  {
    operator_symbol.length = 2;
    operator_symbol.str_value = operator_symbol.bcursor + 2 <= 80
      -> "> "
      #  ["\n", operator_symbol.padding, "> "];
    operator_symbol.ecursor = operator_symbol.bcursor + 2 <= 80
      -> operator_symbol.bcursor + 2
      #  len(operator_symbol.padding) + 2;
  }
| '='
  {
    operator_symbol.length = 2;
    operator_symbol.str_value = operator_symbol.bcursor + 2 <= 80
      -> "= "
      #  ["\n", operator_symbol.padding, "= "];
    operator_symbol.ecursor = operator_symbol.bcursor + 2 <= 80
      -> operator_symbol.bcursor + 2
      #  len(operator_symbol.padding) + 2;
  }
| LE
  {
    operator_symbol.length = 3;
    operator_symbol.str_value = operator_symbol.bcursor + 3 <= 80
      -> "<= "
      #  ["\n", operator_symbol.padding, "<= "];
    operator_symbol.ecursor = operator_symbol.bcursor + 3 <= 80
      -> operator_symbol.bcursor + 3
      #  len(operator_symbol.padding) + 3;
  }
| GE
  {
    operator_symbol.length = 3;
```

```
      operator_symbol.str_value = operator_symbol.bcursor + 3 <= 80
        -> ">= "
        #  ["\n", operator_symbol.padding, ">= "];
      operator_symbol.ecursor = operator_symbol.bcursor + 3 <= 80
        -> operator_symbol.bcursor + 3
        #  len(operator_symbol.padding) + 3;
    }
  | NE
    {
      operator_symbol.length = 3;
      operator_symbol.str_value = operator_symbol.bcursor + 3 <= 80
        -> "~= "
        #  ["\n", operator_symbol.padding, "~= "];
      operator_symbol.ecursor = operator_symbol.bcursor + 3 <= 80
        -> operator_symbol.bcursor + 3
        #  len(operator_symbol.padding) + 3;
    }
  | NLT
    {
      operator_symbol.length = 3;
      operator_symbol.str_value = operator_symbol.bcursor + 3 <= 80
        -> "~< "
        #  ["\n", operator_symbol.padding, "~< "];
      operator_symbol.ecursor = operator_symbol.bcursor + 3 <= 80
        -> operator_symbol.bcursor + 3
        #  len(operator_symbol.padding) + 3;
    }
  | NGT
    {
      operator_symbol.length = 3;
      operator_symbol.str_value = operator_symbol.bcursor + 3 <= 80
        -> "~> "
        #  ["\n", operator_symbol.padding, "~> "];
      operator_symbol.ecursor = operator_symbol.bcursor + 3 <= 80
        -> operator_symbol.bcursor + 3
        #  len(operator_symbol.padding) + 3;
    }
  | NLE
    {
      operator_symbol.length = 4;
      operator_symbol.str_value = operator_symbol.bcursor + 4 <= 80
        -> "~<= "
        #  ["\n", operator_symbol.padding, "~<= "];
      operator_symbol.ecursor = operator_symbol.bcursor + 4 <= 80
        ->  operator_symbol.bcursor + 4
        #  len(operator_symbol.padding) + 4;
    }
  | NGE
    {
      operator_symbol.length = 4;
      operator_symbol.str_value = operator_symbol.bcursor + 4 <= 80
```

```
     -> "~>= "
     #  ["\n", operator_symbol.padding, "~>= "];
   operator_symbol.ecursor = operator_symbol.bcursor + 4 <= 80
     -> operator_symbol.bcursor + 4
     #  len(operator_symbol.padding) + 4;
 }
| EQV
 {
   operator_symbol.length = 3;
   operator_symbol.str_value = operator_symbol.bcursor + 3 <= 80
     -> "== "
     #  ["\n", operator_symbol.padding, "== "];
   operator_symbol.ecursor = operator_symbol.bcursor + 3 <= 80
     -> operator_symbol.bcursor + 3
     #  len(operator_symbol.padding) + 3;
 }
| NEQV
 {
   operator_symbol.length = 4;
   operator_symbol.str_value = operator_symbol.bcursor + 4 <= 80
     -> "~== "
     #  ["\n", operator_symbol.padding, "~== "];
   operator_symbol.ecursor = operator_symbol.bcursor + 4 <= 80
     -> operator_symbol.bcursor + 4
     #  len(operator_symbol.padding) + 4;
 }
| '+'
 {
   operator_symbol.length = 2;
   operator_symbol.str_value = operator_symbol.bcursor + 2 <= 80
     -> "+ "
     #  ["\n", operator_symbol.padding, "+ "];
   operator_symbol.ecursor = operator_symbol.bcursor + 2 <= 80
     -> operator_symbol.bcursor + 2
     #  len(operator_symbol.padding) + 2;
 }
| '-'
 {
   operator_symbol.length = 2;
   operator_symbol.str_value = operator_symbol.bcursor + 2 <= 80
     -> "- "
     #  ["\n", operator_symbol.padding, "- "];
   operator_symbol.ecursor = operator_symbol.bcursor + 2 <= 80
     -> operator_symbol.bcursor + 2
     #  len(operator_symbol.padding) + 2;
 }
| '*'
 {
   operator_symbol.length = 2;
   operator_symbol.str_value = operator_symbol.bcursor + 2 <= 80
     -> "* "
```

```
        # ["\n", operator_symbol.padding, "* "];
      operator_symbol.ecursor = operator_symbol.bcursor + 2 <= 80
        -> operator_symbol.bcursor + 2
        #  len(operator_symbol.padding) + 2;
    }
  | '/'
    {
      operator_symbol.length = 2;
      operator_symbol.str_value = operator_symbol.bcursor + 2 <= 80
        -> "/ "
        #  ["\n", operator_symbol.padding, "/ "];
      operator_symbol.ecursor = operator_symbol.bcursor + 2 <= 80
        -> operator_symbol.bcursor + 2
        #  len(operator_symbol.padding) + 2;
    }
  | MOD
    {
      operator_symbol.length = 4;
      operator_symbol.str_value = operator_symbol.bcursor + 4 <= 80
        -> "MOD "
        #  ["\n", operator_symbol.padding, "MOD "];
      operator_symbol.ecursor = operator_symbol.bcursor + 4 <= 80
        -> operator_symbol.bcursor + 4
        #  len(operator_symbol.padding) + 4;
    }
  | EXP
    {
      operator_symbol.length = 3;
      operator_symbol.str_value = operator_symbol.bcursor + 3 <= 80
        -> "** "
        #  ["\n", operator_symbol.padding, "** "];
      operator_symbol.ecursor = operator_symbol.bcursor + 3 <= 80
        -> operator_symbol.bcursor + 3
        #  len(operator_symbol.padding) + 3;
    }
  | U
    {
      operator_symbol.length = 2;
      operator_symbol.str_value = operator_symbol.bcursor + 2 <= 80
        -> "U "
        #  ["\n", operator_symbol.padding, "U "];
      operator_symbol.ecursor = operator_symbol.bcursor + 2 <= 80
        -> operator_symbol.bcursor + 2
        #  len(operator_symbol.padding) + 2;
    }
  | APPEND
    {
      operator_symbol.length = 3;
      operator_symbol.str_value = operator_symbol.bcursor + 3 <= 80
        -> "|| "
        #  ["\n", operator_symbol.padding, "|| "];
        operator_symbol.ecursor = operator_symbol.bcursor + 3 <= 80
          -> operator_symbol.bcursor + 3
          #  len(operator_symbol.padding) + 3;
      }
    | IN
      {
        operator_symbol.length = 3;
        operator_symbol.str_value = operator_symbol.bcursor + 3 <= 80
          -> "IN "
          #  ["\n", operator_symbol.padding, "IN "];
        operator_symbol.ecursor = operator_symbol.bcursor + 3 <= 80
          -> operator_symbol.bcursor + 3
          #  len(operator_symbol.padding) + 3;
      }
    | RANGE
      {
        operator_symbol.length = 3;
        operator_symbol.str_value = operator_symbol.bcursor + 3 <= 80
          -> ".. "
          #  ["\n", operator_symbol.padding, ".. "];
        operator_symbol.ecursor = operator_symbol.bcursor + 3 <= 80
          -> operator_symbol.bcursor + 3
          #  len(operator_symbol.padding) + 3;
      }
    | '.'
      {
        operator_symbol.length = 2;
        operator_symbol.str_value = operator_symbol.bcursor + 2 <= 80
          -> ". "
          #  ["\n", operator_symbol.padding, ". "];
        operator_symbol.ecursor = operator_symbol.bcursor + 2 <= 80
          -> operator_symbol.bcursor + 2
          #  len(operator_symbol.padding) + 2;
      }
    | '['
      {
        operator_symbol.length = 2;
        operator_symbol.str_value = operator_symbol.bcursor + 2 <= 80
          -> "[ "
          #  ["\n", operator_symbol.padding, "[ "];
        operator_symbol.ecursor = operator_symbol.bcursor + 2 <= 80
          -> operator_symbol.bcursor + 2
          #  len(operator_symbol.padding) + 2;
      }
    ;

comment
    : COMMENT comment
      {
        comment[1].str_value = comment[1].bcursor == 0
          -> [COMMENT.%text,comment[2].str_value]
      #  comment[1].bcursor + 2 + len(COMMENT.%text) <= 80
      -> ["  ", COMMENT.%text, comment[2].str_value]
      #  ["\n",COMMENT.%text, comment[2].str_value];
     comment[2].bcursor = 0;
     comment[1].length = len(COMMENT.%text) + comment[2].length;
  }
 |                    %prec SEMI
  {
    comment.str_value = "";
    comment.length = 0;
  }
 ;
```

## APPENDIX C

SAMPLE INPUT 1

```
FUNCTION square_root {precision: real} WHERE precision > 0.0
  MESSAGE(x: real)
  WHEN x >= 0.0
  REPLY(y: real)
  WHERE y >= 0.0 & approximates(y * y, x)
  OTHERWISE REPLY EXCEPTION imaginary_square_root
  CONCEPT approximates(r1 r2: real)
-- True if r1 is a sufficiently accurate approximation of r2.
-- The precision is relative rather than absolute.
  VALUE(b: boolean)
  WHERE b <=> abs((r1 - r2) / r2) <= precision
END
```

SAMPLE OUTPUT 1

```
FUNCTION square_root(precision : real)
   WHERE precision > 0.0

   MESSAGE (x : real)
      WHEN x >= 0.0
         REPLY (y : real)
            WHERE y >= 0.0 & approximates(y * y, x)
      OTHERWISE
         REPLY EXCEPTION imaginary_square_root

   CONCEPT approximates(r1 r2 : real)
-- True if r1 is a sufficiently accurate approximation of r2.
-- The precision is relative rather than absolute.

      VALUE (b : boolean)
      WHERE b <=> abs((r1 - r2) / r2) <= precision
END
```

SAMPLE INPUT 2

```
TYPE rational   INHERIT equality{rational}
 MODEL(num den: integer)
 INVARIANT ALL(r: rational :: r.den ~= 0)
 MESSAGE ratio(num den: integer)
 WHEN den ~= 0       REPLY(r: rational)
 WHERE r.num = num, r.den = den
 OTHERWISE REPLY EXCEPTION zero_denominator
  MESSAGE add(x y: rational) OPERATOR +
  REPLY(r: rational)
  WHERE r.num = x.num * y.den + y.num * x.den, r.den = x.den * y.den

  MESSAGE multiply(x y: rational) OPERATOR *
  REPLY(r: rational)
  WHERE r.num = x.num * y.num, r.den = x.den * y.den

 MESSAGE equal(x y: rational) OPERATOR =
 REPLY(b: boolean)
 WHERE b <=> (x.num * y.den = y.num * x.den)
END
```

SAMPLE OUTPUT 2

```
TYPE rational
   INHERIT equality{rational}

   MODEL (num den : integer)
      INVARIANT ALL(r : rational :: r.den ~= 0)

   MESSAGE ratio(num den : integer)
      WHEN den ~= 0
         REPLY (r : rational)
            WHERE r.num = num, r.den = den
      OTHERWISE
         REPLY EXCEPTION zero_denominator

   MESSAGE add(x y : rational) OPERATOR +
      REPLY (r : rational)
         WHERE r.num = x.num * y.den + y.num * x.den, r.den = x.den * y.den

   MESSAGE multiply(x y : rational) OPERATOR *
      REPLY (r : rational)
         WHERE r.num = x.num * y.num, r.den = x.den * y.den

   MESSAGE equal(x y : rational) OPERATOR =
      REPLY (b : boolean)
         WHERE b <=> (x.num * y.den = y.num * x.den)
END
```

SAMPLE INPUT 3

```
MACHINE airline_manager_interface   STATE (?)   INVARIANT ?   INITIALLY ?

MESSAGE add_flight(i: flight_id, price: money, origin destination: airport,
 departure arrival: time, capacity: nat)   WHEN ? -- new flight       REPLY done
 TRANSITION ? -- add flight
OTHERWISE REPLY EXCEPTION flight_exists

 MESSAGE drop_flight(i: flight_id)     WHEN ? -- flight exists   REPLY done
 TRANSITION ? -- remove flight  OTHERWISE REPLY EXCEPTION no_such_flight
MESSAGE new_fare(i: flight_id, price: money)
WHEN ? -- flight exists       REPLY done       TRANSITION ? -- change fare
OTHERWISE REPLY EXCEPTION no_such_flight
END
```

SAMPLE OUTPUT 3

```
MACHINE airline_manager_interface

   STATE (?)
      INVARIANT ?
      INITIALLY ?

   MESSAGE add_flight(i : flight_id, price : money, origin destination : airport,
                      departure arrival : time, capacity : nat)
      WHEN ?  -- new flight

         REPLY done
         TRANSITION ?  -- add flight

      OTHERWISE
         REPLY EXCEPTION flight_exists

   MESSAGE drop_flight(i : flight_id)
      WHEN ?  -- flight exists

         REPLY done
         TRANSITION ?  -- remove flight

      OTHERWISE
         REPLY EXCEPTION no_such_flight

   MESSAGE new_fare(i : flight_id, price : money)
      WHEN ?  -- flight exists

         REPLY done
         TRANSITION ?  -- change fare

      OTHERWISE
         REPLY EXCEPTION no_such_flight
END
```

SAMPLE INPUT 4

```
MACHINE sender   STATE(data: sequence{block})   INVARIANT true   INITIALLY true
MESSAGE send(file: sequence{block})   WHEN length(file) > 0
SEND first(b: block) TO receiver WHERE b = file[1]
TRANSITION data = file
OTHERWISE REPLY EXCEPTION empty_file

MESSAGE echo(b: block)
WHEN b = *data[1] & length(*data) > 1
SEND next(b1: block) TO receiver WHERE b1 = data[1]
TRANSITION *data = b || data
WHEN b = *data[1] & length(*data) = 1
SEND done TO receiver
SEND done TO sender
TRANSITION data = [ ]
OTHERWISE SEND retransmit(b2: block) TO receiver WHERE b2 = data[1]
MESSAGE done
TRANSACTION transfer = send ; DO echo OD ; done
END
```

SAMPLE OUTPUT 4

```
MACHINE sender

   STATE (data : sequence{block})
      INVARIANT true
      INITIALLY true

   MESSAGE send(file : sequence{block})
      WHEN length(file) > 0
         SEND first(b : block)
            TO receiver
            WHERE b = file[1]
         TRANSITION data = file
      OTHERWISE
         REPLY EXCEPTION empty_file

   MESSAGE echo(b : block)
      WHEN b = *data[1] & length(*data) > 1
         SEND next(b1 : block)
            TO receiver
            WHERE b1 = data[1]
         TRANSITION *data = b || data
      WHEN b = *data[1] & length(*data) = 1
         SEND done
            TO receiver
         SEND done
            TO sender
         TRANSITION data = []
      OTHERWISE
         SEND retransmit(b2 : block)
            TO receiver
            WHERE b2 = data[1]

   MESSAGE done

   TRANSACTION transfer = send; DO echo OD; done
END
```

SAMPLE INPUT 5

```
TYPE char   INHERIT equality{char}   INHERIT total_order{char}

MODEL(code: nat)     -- ASCII codes
INVARIANT ALL(c: char :: 0 <= c.code <= 127)

MESSAGE create(n: nat) -- literal 'a' = create(97) and so on
WHEN 0 <= n <= 127  REPLY(c: char) WHERE c.code = n
OTHERWISE REPLY EXCEPTION illegal_code

MESSAGE ordinal(c: char)     REPLY(n: nat)     WHERE n = c.code

MESSAGE equal(c1 c2: char)     REPLY(b: boolean)     WHERE b <=> (c1.code = c2.code)

MESSAGE less(c1 c2: char)     REPLY(b: boolean)     WHERE b <=> (c1.code < c2.code)

MESSAGE letter(c: char)  REPLY(b: boolean)  WHERE b <=> c IN ['a' .. 'z'] ¦ c IN
['A' .. 'Z']

MESSAGE digit(c: char)     REPLY(b: boolean)  WHERE b <=> c IN ['0' .. '9']
END
```

SAMPLE OUTPUT 5

```
TYPE char
   INHERIT equality{char}
   INHERIT total_order{char}

   MODEL (code : nat)  -- ASCII codes

      INVARIANT ALL(c : char :: 0 <= c.code <= 127)

   MESSAGE create(n : nat)  -- literal 'a' = create(97) and so on

      WHEN 0 <= n <= 127
         REPLY (c : char)
            WHERE c.code = n
      OTHERWISE
         REPLY EXCEPTION illegal_code

   MESSAGE ordinal(c : char)
      REPLY (n : nat)
         WHERE n = c.code

   MESSAGE equal(c1 c2 : char)
      REPLY (b : boolean)
         WHERE b <=> (c1.code = c2.code)

   MESSAGE less(c1 c2 : char)
      REPLY (b : boolean)
         WHERE b <=> (c1.code < c2.code)

   MESSAGE letter(c : char)
      REPLY (b : boolean)
         WHERE b <=> c IN ['a' .. 'z'] ¦ c IN ['A' .. 'Z']

   MESSAGE digit(c : char)
      REPLY (b : boolean)
         WHERE b <=> c IN ['0' .. '9']
END
```

SAMPLE INPUT 6

```
MACHINE inventory     -- assumes that shipping and supplier are other modules.
  STATE(stock: map{item, integer})
  INVARIANT ALL(i: item :: stock[i] >= 0)
  INITIALLY ALL(i: item :: stock[i] = 0)

  MESSAGE receive(i: item, q: integer)       -- Process a shipment from a supplier.
    WHEN q > 0
      TRANSITION stock[i] = *stock[i] + q
        -- Delayed responses to backorders are not shown here.
OTHERWISE REPLY EXCEPTION empty_shipment

MESSAGE order(io: item, qo: integer)
      -- Process an order from a customer.
    WHEN 0 < qo <= stock[io]
      SEND ship(is: item, qs: integer) TO shipping   WHERE is = io, qs = qo
      TRANSITION stock[io] + qo = *stock[io]
    WHEN 0 < qo > stock[io]   SEND ship(is: item, qs: integer) TO shipping
        WHERE is = io, qs = stock[io]
SEND back_order(ib: item, qb: integer) TO supplier
        WHERE ib = io, qb + qs = qo
      TRANSITION stock[io] = 0
    OTHERWISE REPLY EXCEPTION empty_order
END
```

SAMPLE OUTPUT 6

```
MACHINE inventory  -- assumes that shipping and supplier are other modules.


   STATE (stock : map{item, integer})
      INVARIANT ALL(i : item :: stock[i] >= 0)
      INITIALLY ALL(i : item :: stock[i] = 0)

   MESSAGE receive(i : item, q : integer)
-- Process a shipment from a supplier.

      WHEN q > 0
         TRANSITION stock[i] = *stock[i] + q
-- Delayed responses to backorders are not shown here.

      OTHERWISE
         REPLY EXCEPTION empty_shipment

   MESSAGE order(io : item, qo : integer)  -- Process an order from a customer.

      WHEN 0 < qo <= stock[io]
         SEND ship(is : item, qs : integer)
            TO shipping
            WHERE is = io, qs = qo
         TRANSITION stock[io] + qo = *stock[io]
      WHEN 0 < qo > stock[io]
         SEND ship(is : item, qs : integer)
            TO shipping
            WHERE is = io, qs = stock[io]
         SEND back_order(ib : item, qb : integer)
            TO supplier
            WHERE ib = io, qb + qs = qo
         TRANSITION stock[io] = 0
      OTHERWISE
         REPLY EXCEPTION empty_order
END
```

SAMPLE INPUT 7

```
VIRTUAL TYPE number{t: type}
INHERIT equality{t}
EXPORT commutative associative distributive
MODEL
INVARIANT true
MESSAGE zero
REPLY(n: t)
MESSAGE one
REPLY(n: t)
MESSAGE plus(n1 n2: t) OPERATOR +
REPLY(i3: t)
WHERE ALL(n: t :: n + zero = n), commutative(plus), associative(plus)
MESSAGE times(n1 n2: t) OPERATOR *
REPLY(i3: t)
WHERE ALL(n: t :: n * zero = zero),
ALL(n: t :: n * one = n),
commutative(times), associative(times), distributive(plus, times)
CONCEPT commutative(f: FUNCTION)
VALUE(b: boolean)
WHERE b <=> domain(f) = [t, t] & range(f) = t
& ALL(x y: t :: f(x, y) = f(y, z))
CONCEPT associative(f: FUNCTION)
VALUE(b: boolean)
WHERE b <=> domain(f) = [t, t] & range(f) = t
& ALL(x y z: t :: f(x, f(y, z)) = f(f(x, y), z))
CONCEPT distributive(f g: FUNCTION)
VALUE(b: boolean)
WHERE b <=> domain(f) = [t, t] & range(f) = t
& domain(g) = [t, t] & range(g) = t
& ALL(x y z: t :: g(x, f(y, z)) = f(g(x, y), g(x, z)))
END
```

SAMPLE OUTPUT 7

```
VIRTUAL TYPE number{t : type}
   INHERIT equality{t}
   EXPORT commutative associative distributive

   MODEL
      INVARIANT true

   MESSAGE zero
      REPLY (n : t)

   MESSAGE one
      REPLY (n : t)

   MESSAGE plus(n1 n2 : t) OPERATOR +
      REPLY (l3 : t)
         WHERE ALL(n : t :: n + zero = n), commutative(plus), associative(plus)

   MESSAGE times(n1 n2 : t) OPERATOR *
      REPLY (l3 : t)
         WHERE ALL(n : t :: n * zero = zero), ALL(n : t :: n * one = n),
               commutative(times), associative(times), distributive(plus, times)

   CONCEPT commutative(f : FUNCTION)
      VALUE (b : boolean)
      WHERE b
            <=> domain(f) = [t, t] & range(f) = t
                & ALL(x y : t :: f(x, y) = f(y, z))

   CONCEPT associative(f : FUNCTION)
      VALUE (b : boolean)
      WHERE b
            <=> domain(f) = [t, t] & range(f) = t
                & ALL(x y z : t :: f(x, f(y, z)) = f(f(x, y), z))

   CONCEPT distributive(f g : FUNCTION)
      VALUE (b : boolean)
      WHERE b
            <=> domain(f) = [t, t] & range(f) = t & domain(g) = [t, t]
                & range(g) = t
                & ALL(x y z : t :: g(x, f(y, z)) = f(g(x, y), g(x, z)))
END
```

SAMPLE INPUT 8

```
MACHINE airline_manager
INHERIT format
IMPORT edit flight_id money airport time FROM travel_agent

STATE
INVARIANT true
INITIALLY true

MESSAGE (command: string)
      -- command from airline manager's keyboard
WHEN add_flight(edit(command), i, price, origin, destination,
                    departure, arrival, capacity)
SEND add_flight(i: flight_id, price: money, origin destination: airport,
                      departure arrival: time, capacity: nat)
TO airline_reservation_system
WHEN drop_flight(edit(command), i)
SEND drop_flight(i: flight_id)
TO airline_reservation_system
WHEN new_fare(edit(command), i, price)
SEND new_fare(i: flight_id, price: money)
TO airline_reservation_system
OTHERWISE REPLY(s: string)
WHERE s = "command not recognized"

  -- normal responses

MESSAGE done     SEND(s: string) TO display WHERE s = "done"

  -- error messages

  -- input formats

CONCEPT add_flight(command: string, i: flight_id, p: money,
                     o d: airport, dep arr: time, cn: nat)
VALUE(b: boolean)
WHERE b <=> SOME(cap: string SUCH THAT cn = nat(cap) :: command = list("a", i, p, o, d,
  dep, arr, cap) )

 CONCEPT drop_flight(command: string, i: flight_id)
 VALUE(b: boolean)
 WHERE b <=> command = list("d", i)

 CONCEPT new_fare(command: string, i: flight_id, p: money)
    VALUE(b: boolean)
    WHERE b <=> command = list("n", i, p)
END
```

SAMPLE OUTPUT 8

```
MACHINE airline_manager
   INHERIT format
   IMPORT edit flight_id money airport time
      FROM travel_agent

   STATE
      INVARIANT true
      INITIALLY true

   MESSAGE (command : string)  -- command from airline manager's keyboard

      WHEN add_flight(edit(command), i, price, origin, destination, departure,
                      arrival, capacity)
         SEND add_flight(i : flight_id, price : money, origin destination
                         : airport, departure arrival : time, capacity : nat)
            TO airline_reservation_system
      WHEN drop_flight(edit(command), i)
         SEND drop_flight(i : flight_id)
            TO airline_reservation_system
      WHEN new_fare(edit(command), i, price)
         SEND new_fare(i : flight_id, price : money)
            TO airline_reservation_system
      OTHERWISE
         REPLY (s : string)
            WHERE s = "command not recognized"  -- normal responses


   MESSAGE done
      SEND (s : string)
         TO display
         WHERE s = "done"  -- error messages
-- input formats


   CONCEPT add_flight(command : string, i : flight_id, p : money, o d : airport,
                      dep arr : time, cn : nat)
      VALUE (b : boolean)
      WHERE b
            <=> SOME(cap : string SUCH THAT cn = nat(cap)
                   :: command = list("a", i, p, o, d, dep, arr, cap))

   CONCEPT drop_flight(command : string, i : flight_id)
      VALUE (b : boolean)
      WHERE b <=> command = list("d", i)

   CONCEPT new_fare(command : string, i : flight_id, p : money)
      VALUE (b : boolean)
      WHERE b <=> command = list("n", i, p)
```

```
END
```

SAMPLE INPUT 9

```
TYPE union{$s: type} WHERE distinct(identifiers(s))
  -- A union type is a tagged disjoint union of a set of types.
  -- Two union types are the same iff they have the same
  -- actual parameters WITH THE SAME FIELD NAMES.
  -- identifiers(s) is the sequence of identifiers used as
  -- field names in the actual parameter list.

INHERIT equality{union{$s}}
IMPORT identifiers identifier FROM field_names IMPORT distinct FROM sequence{identifier}

MODEL(tag: identifier, value: any)
INVARIANT ALL(u: union{$s} :: u.tag IN identifiers(s)),
ALL(u: union{$s} :: u.value IN type_of(u.tag))

MESSAGE create{id: identifier} WHERE id IN identifiers(s)              (x: any)
    -- literal {t :: v} = create{t}(v)
WHEN x IN type_of(id)       REPLY(u: union{$s}) WHERE u.tag = id & u.value = x
OTHERWISE REPLY EXCEPTION type_error

MESSAGE is{id: identifier} WHERE id IN identifiers(s)  (u: oneof{$s})
REPLY(b: boolean) WHERE b <=> u.tag = id
    -- Check if you have a given variant.

MESSAGE get{id: identifier} WHERE id IN identifiers(s)  (u: oneof{$s}) OPERATOR .
    WHEN u.tag = id
      CHOOSE(rt: type SUCH THAT rt = type_of(id))
      REPLY(x: rt) WHERE x = u.value
    OTHERWISE REPLY EXCEPTION type_error
    -- Extract the value assuming a given variant,
    -- succeeds only if the tag matches the assumed variant.

MESSAGE equal(u1 u2: union{$s})
    REPLY(b: boolean)
    WHERE b <=> u1.tag = u2.tag & u1.value = u2.value

  CONCEPT type_of(id: identifier)
      WHERE id IN identifiers(s)
    VALUE(t: type)
    -- The type corresponding to the id in the formal parameter list s.
      WHERE SOME(n: nat :: t = s[n] & id = identifiers(s)[n])
END
```

SAMPLE OUTPUT 9

```
TYPE union{$s: type}
   WHERE distinct(identifiers(s))
-- A union type is a tagged disjoint union of a set of types.
-- Two union types are the same iff they have the same
-- actual parameters WITH THE SAME FIELD NAMES.
-- identifiers(s) is the sequence of identifiers used as
-- field names in the actual parameter list.

   INHERIT equality{union{$s}}
   IMPORT identifiers identifier
      FROM field_names
   IMPORT distinct
      FROM sequence{identifier}

   MODEL (tag : identifier, value : any)
      INVARIANT ALL(u : union{$s} :: u.tag IN identifiers(s)),
                ALL(u : union{$s} :: u.value IN type_of(u.tag))

   MESSAGE create{id : identifier}
      WHERE id IN identifiers(s)
      (x : any)  -- literal {t :: v} = create{t}(v)

      WHEN x IN type_of(id)
         REPLY (u : union{$s})
            WHERE u.tag = id & u.value = x
      OTHERWISE
         REPLY EXCEPTION type_error

   MESSAGE is{id : identifier}
      WHERE id IN identifiers(s)
      (u : oneof{$s})
      REPLY (b : boolean)
         WHERE b <=> u.tag = id  -- Check if you have a given variant.


   MESSAGE get{id : identifier}
      WHERE id IN identifiers(s)
      (u : oneof{$s}) OPERATOR .
      WHEN u.tag = id
         CHOOSE(rt : type SUCH THAT rt = type_of(id))
         REPLY (x : rt)
            WHERE x = u.value
      OTHERWISE
         REPLY EXCEPTION type_error
-- Extract the value assuming a given variant,
-- succeeds only if the tag matches the assumed variant.
```

```
   MESSAGE equal(u1 u2 : union{$s})
      REPLY (b : boolean)
         WHERE b <=> u1.tag = u2.tag & u1.value = u2.value

   CONCEPT type_of(id : identifier)
      WHERE id IN identifiers(s)
      VALUE (t : type)
-- The type corresponding to the id in the formal parameter list s.

      WHERE SOME(n : nat :: t = s[n] & id = identifiers(s)[n])
END
```

SAMPLE INPUT 10

```
MACHINE travel_agent_interface   IMPORT flight FROM airline_manager_interface

  STATE(reservations: set{reservation}, schedule: map{flight_id, flight})
  INVARIANT existing_flights(reservations), no_overbooking(reservations)
  INITIALLY reservations = { }

  MESSAGE find_flights(origin destination: airport)  -- G1.1.1, G1.1.2
  REPLY flights(s: set{flight})
    WHERE ALL(f: flight :: f IN s <=> f IN range(schedule)& f.origin = origin
                                      & f.destination = destination )
    -- flights from the origin to the destination
  MESSAGE reserve(i: flight_id, d: date, p: passenger)  -- G1.2
    WHEN i IN schedule & bookings(i, d) < schedule[i].capacity
        & ~([id:: i, d:: d, p:: p] IN *reservations) -- seat available
      REPLY done
      TRANSITION reservations = *reservations U {[id:: i, d:: d, p:: p]}
        -- add reservation
    WHEN [id:: i, d:: d, p:: p] IN *reservations
      REPLY EXCEPTION reservation_exists
    WHEN ~(i IN schedule) -- unknown flight
      REPLY EXCEPTION no_such_flight
    OTHERWISE REPLY EXCEPTION no_seat
  MESSAGE cancel(i: flight_id, d: date, p: passenger)  -- G1.3
    WHEN i IN schedule & [id:: i, d:: d, p:: p] IN reservations
        -- reservation found
      REPLY done
      TRANSITION reservations = *reservations - {[id:: i, d:: d, p:: p]}
        -- remove reservation
    WHEN ~(i IN schedule) -- unknown flight
      REPLY EXCEPTION no_such_flight
    OTHERWISE REPLY EXCEPTION no_reservation
  CONCEPT existing_flights(s: set{reservation})
    VALUE(b: boolean)
    WHERE ALL(r: reservation SUCH THAT r IN s :: r.flight_id IN schedule)
  CONCEPT no_overbooking(s: set{reservation})
    VALUE(b: boolean)
    WHERE ALL(i: flight_id, d: date SUCH THAT i IN schedule
              :: bookings(i, d) <= schedule[i].capacity )
  CONCEPT bookings(i: flight_id, d: date)
    VALUE(n: nat)
    WHERE n = NUMBER(r: reservation
                     SUCH THAT r IN reservations & r.id = i & r.d = d  :: r )

  CONCEPT reservation: type
    WHERE reservation = tuple{id:: flight_id, d:: date, p:: passenger}
END
```

SAMPLE OUTPUT 10

```
MACHINE travel_agent_interface
   IMPORT flight
      FROM airline_manager_interface

   STATE (reservations : set{reservation}, schedule : map{flight_id, flight})
      INVARIANT existing_flights(reservations), no_overbooking(reservations)
      INITIALLY reservations = {}

   MESSAGE find_flights(origin destination : airport)  -- G1.1.1, G1.1.2

      REPLY flights(s : set{flight})
         WHERE ALL(f : flight
                   :: f IN s
                      <=> f IN range(schedule) & f.origin = origin
                          & f.destination = destination)
-- flights from the origin to the destination


   MESSAGE reserve(i : flight_id, d : date, p : passenger)  -- G1.2

      WHEN i IN schedule & bookings(i, d) < schedule[i].capacity
           & ~([id :: i, d :: d, p :: p] IN *reservations)  -- seat available

         REPLY done
         TRANSITION reservations = *reservations U {[id :: i, d :: d, p :: p]}
-- add reservation

      WHEN [id :: i, d :: d, p :: p] IN *reservations
         REPLY EXCEPTION reservation_exists
      WHEN ~(i IN schedule)  -- unknown flight

         REPLY EXCEPTION no_such_flight
      OTHERWISE
         REPLY EXCEPTION no_seat

   MESSAGE cancel(i : flight_id, d : date, p : passenger)  -- G1.3

      WHEN i IN schedule & [id :: i, d :: d, p :: p] IN reservations
-- reservation found

         REPLY done
         TRANSITION reservations = *reservations - {[id :: i, d :: d, p :: p]}
-- remove reservation

      WHEN ~(i IN schedule)  -- unknown flight

         REPLY EXCEPTION no_such_flight
      OTHERWISE
```

```
        REPLY EXCEPTION no_reservation

   CONCEPT existing_flights(s : set{reservation})
      VALUE (b : boolean)
      WHERE ALL(r : reservation SUCH THAT r IN s :: r.flight_id IN schedule)

   CONCEPT no_overbooking(s : set{reservation})
      VALUE (b : boolean)
      WHERE ALL(i : flight_id, d : date SUCH THAT i IN schedule
                 :: bookings(i, d) <= schedule[i].capacity)

   CONCEPT bookings(i : flight_id, d : date)
      VALUE (n : nat)
      WHERE n
            = NUMBER(r : reservation
                     SUCH THAT r IN reservations & r.id = i & r.d = d :: r)

   CONCEPT reservation: type
      WHERE reservation = tuple{id :: flight_id, d :: date, p :: passenger}
END
```

SAMPLE INPUT 11

```
TYPE real
  INHERIT number{real}
  INHERIT total_order{real}

  MODEL
  INVARIANT true

  MESSAGE rational_to_real(r1: rational)
    -- type conversion operation for mixed mode arithmetic
    REPLY(r2: real)
    WHERE rational_to_real(zero) = zero, rational_to_real(one) = one,
      ALL(x y: rational :: rational_to_real(x - y) = rational_to_real(x) -
 rational_to_real(y)),
      ALL(x y: rational SUCH THAT y ~= zero :: rational_to_real(x / y) =
 rational_to_real(x) / rational_to_real(y))

  MESSAGE integer_to_real(i: integer)
    -- type conversion operation for mixed mode arithmetic
    REPLY(r: real)
    WHERE r = rational_to_real(integer_to_rational@rational(i))

  MESSAGE nat_to_real(n: nat)
    -- type conversion operation for mixed mode arithmetic
    REPLY(r: real)
    WHERE r = rational_to_real(nat_to_rational@rational(n))

  MESSAGE rational(r: real)
    REPLY(b: boolean)
    WHERE SOME(i: integer :: r = rational_to_real(i))

  MESSAGE integral(r: real)
    REPLY(b: boolean)
    WHERE SOME(i: integer :: r = integer_to_real(i))

  MESSAGE nat(r: real)
    REPLY(b: boolean)
    WHERE SOME(n: nat :: r = nat_to_real(n))

  MESSAGE zero
    -- literal 0.0 = zero
    REPLY(l: real)

  MESSAGE one
    -- literal 1.0 = one
    REPLY(l: real)

  MESSAGE minus(r1: real) OPERATOR -
    REPLY(r2: real)
    WHERE r2 = 0.0 - r1

  MESSAGE plus(r1 r2: real) OPERATOR +
    REPLY(r3: real)

  MESSAGE difference(r1 r2: real) OPERATOR -
    REPLY(r3: real)
    WHERE r1 = r2 + r3

  MESSAGE times(r1 r2: real) OPERATOR *
    REPLY(r3: real)

  MESSAGE quotient(r1 r2: real) OPERATOR /
    WHEN r2 ~= zero
      REPLY(r3: real)
      WHERE r1 = r2 * r3
    OTHERWISE REPLY EXCEPTION divide_by_zero

  MESSAGE remainder(r1 r2: real) OPERATOR \ MOD
    WHEN r2 ~= zero
      REPLY(r: real)
      WHERE SOME(q: real :: r1 = q * r2 + r & abs(r2) > r >= zero & integral(q))
    OTHERWISE REPLY EXCEPTION divide_by_zero

  MESSAGE expt(r1 r2: real) OPERATOR **
    WHEN (r1 = 0.0 & r2 <= 0.0) ¦ (r1 < 0.0 & ~integral(r2))
      REPLY EXCEPTION undefined_expt
    OTHERWISE REPLY(r3: real)
      WHERE ALL(r: real :: r ** 1.0 = r), ALL(r: real SUCH THAT r > 0.0 :: 0.0

** r = 0.0),
        ALL(r x y: real SUCH THAT r > 0.0 ¦ r < 0.0 & integral(x) & integral(y)
            :: r ** (x + y) = (r ** x) * (r ** y) )

  MESSAGE equal(r1 r2: real)
    REPLY(b: boolean)

  MESSAGE less(r1 r2: real)
    REPLY(b: boolean)
    WHERE ALL(x y: rational :: rational_to_real(x) < rational_to_real(y) <=> x < y),
      ALL(x y z: real :: x + y < x + z <=> y < z),
      ALL(x y z: real SUCH THAT x > 0.0 :: x * y < x * z <=> y < z),
      ALL(x y z: real SUCH THAT x < 0.0 :: x * y < x * z <=> y > z)
END
```

SAMPLE OUTPUT 11

```
TYPE real
   INHERIT number{real}
   INHERIT total_order{real}

   MODEL
      INVARIANT true

   MESSAGE rational_to_real(r1 : rational)
-- type conversion operation for mixed mode arithmetic

      REPLY (r2 : real)
         WHERE rational_to_real(zero) = zero, rational_to_real(one) = one,
               ALL(x y : rational
                   :: rational_to_real(x - y)
                      = rational_to_real(x) - rational_to_real(y)),
               ALL(x y : rational SUCH THAT y ~= zero
                   :: rational_to_real(x / y)
                      = rational_to_real(x) / rational_to_real(y))

   MESSAGE integer_to_real(i : integer)
-- type conversion operation for mixed mode arithmetic

      REPLY (r : real)
         WHERE r = rational_to_real(integer_to_rational@rational(i))

   MESSAGE nat_to_real(n : nat)
-- type conversion operation for mixed mode arithmetic

      REPLY (r : real)
         WHERE r = rational_to_real(nat_to_rational@rational(n))

   MESSAGE rational(r : real)
      REPLY (b : boolean)
         WHERE SOME(i : integer :: r = rational_to_real(i))

   MESSAGE integral(r : real)
      REPLY (b : boolean)
         WHERE SOME(i : integer :: r = integer_to_real(i))

   MESSAGE nat(r : real)
      REPLY (b : boolean)
         WHERE SOME(n : nat :: r = nat_to_real(n))

   MESSAGE zero  -- literal 0.0 = zero

      REPLY (i : real)

   MESSAGE one  -- literal 1.0 = one
   REPLY (i : real)

MESSAGE minus(r1 : real) OPERATOR -
   REPLY (r2 : real)
      WHERE r2 = 0.0 - r1

MESSAGE plus(r1 r2 : real) OPERATOR +
   REPLY (r3 : real)

MESSAGE difference(r1 r2 : real) OPERATOR -
   REPLY (r3 : real)
      WHERE r1 = r2 + r3

MESSAGE times(r1 r2 : real) OPERATOR *
   REPLY (r3 : real)

MESSAGE quotient(r1 r2 : real) OPERATOR /
   WHEN r2 ~= zero
      REPLY (r3 : real)
         WHERE r1 = r2 * r3
   OTHERWISE
      REPLY EXCEPTION divide_by_zero

MESSAGE remainder(r1 r2 : real) OPERATOR MOD MOD
   WHEN r2 ~= zero
      REPLY (r : real)
         WHERE SOME(q : real
                    :: r1 = q * r2 + r & abs(r2) > r >= zero & integral(q))
   OTHERWISE
      REPLY EXCEPTION divide_by_zero

MESSAGE expt(r1 r2 : real) OPERATOR **
   WHEN (r1 = 0.0 & r2 <= 0.0) ¦ (r1 < 0.0 & ~integral(r2))
      REPLY EXCEPTION undefined_expt
   OTHERWISE
      REPLY (r3 : real)
         WHERE ALL(r : real :: r ** 1.0 = r),
               ALL(r : real SUCH THAT r > 0.0 :: 0.0 ** r = 0.0),
               ALL(r x y : real
                   SUCH THAT r > 0.0 ¦ r < 0.0 & integral(x) & integral(y)
                   :: r ** (x + y) = (r ** x) * (r ** y))

MESSAGE equal(r1 r2 : real)
   REPLY (b : boolean)

MESSAGE less(r1 r2 : real)
   REPLY (b : boolean)
      WHERE ALL(x y : rational
                :: rational_to_real(x) < rational_to_real(y) <=> x < y),
            ALL(x y z : real :: x + y < x + z <=> y < z),
          ALL(x y z : real SUCH THAT x > 0.0 :: x * y < x * z <=> y < z),
          ALL(x y z : real SUCH THAT x < 0.0 :: x * y < x * z <=> y > z)
END
```

SAMPLE INPUT 12

```
TYPE complex   INHERIT number{complex}
  MODEL(re im: real)
  INVARIANT true
  MESSAGE real_to_complex(r: real)
    -- type conversion operation for mixed mode arithmetic
    REPLY(c: complex)
    WHERE c.re = r, c.im = 0.0
MESSAGE rational_to_complex(r: rational)
    -- type conversion operation for mixed mode arithmetic
    REPLY(c: complex)
    WHERE c = real_to_complex(rational_to_real@real(r))
MESSAGE integer_to_complex(i: integer)
    -- type conversion operation for mixed mode arithmetic
    REPLY(c: complex)
    WHERE c = real_to_complex(integer_to_real@real(i))
MESSAGE nat_to_complex(n: nat)
    -- type conversion operation for mixed mode arithmetic
    REPLY(c: complex)
    WHERE c = real_to_complex(nat_to_real@real(n))

MESSAGE real(c: complex)
    REPLY(b: boolean)
    WHERE b <=> c.im = 0.0

MESSAGE imaginary(c: complex)
    REPLY(b: boolean)
    WHERE b <=> c.re = 0.0

MESSAGE rational(c: complex)
    REPLY(b: boolean)
    WHERE SOME(r: rational :: c = rational_to_complex(r))

MESSAGE integral(c: complex)
    REPLY(b: boolean)
    WHERE SOME(i: integer :: c = integer_to_complex(i))

MESSAGE nat(c: complex)
    REPLY(b: boolean)
    WHERE SOME(n: nat :: c = nat_to_complex(n))

  MESSAGE zero
    REPLY(c: complex)
    WHERE c = real_to_complex(0.0)

MESSAGE one
    REPLY(c: complex)
    WHERE c = real_to_complex(1.0)
MESSAGE i
```

```
    REPLY(c: complex)
    WHERE i * i = - one

MESSAGE conjugate(c1: complex)
    REPLY(c2: complex)
    WHERE c2.re = c1.re, c2.im = -c1.im

MESSAGE magnitude(c1: complex)
    REPLY(r: real)
    WHERE r = (c.re ** 2 + c.im ** 2) ** 0.5

MESSAGE minus(c1: complex) OPERATOR -
    REPLY(c2: complex)
    WHERE c2 = 0.0 - c1

MESSAGE plus(c1 c2: complex) OPERATOR +
    REPLY(c3: complex)
    WHERE c3.re = c1.re + c2.re, c3.im = c1.im + c2.im

MESSAGE difference(c1 c2: complex) OPERATOR -
    REPLY(c3: complex)
    WHERE c1 = c2 + c3

MESSAGE times(c1 c2: complex) OPERATOR *
    REPLY(c3: complex)
    WHERE c3.re = c1.re * c2.re - c1.im * c2.im,
      c3.im = c1.re * c2.im + c1.im * c2.re

MESSAGE quotient(c1 c2: complex) OPERATOR /
WHEN c2 ~= zero
REPLY(c3: complex)
WHERE c1 = c2 * c3
OTHERWISE REPLY EXCEPTION divide_by_zero

MESSAGE expt(c1 c2: complex) OPERATOR **
    WHEN (c1 = zero & c2.re <= 0.0)
      REPLY EXCEPTION undefined_expt
    OTHERWISE REPLY(c3: complex)
      WHERE ALL(c: complex :: c ** one = c), ALL(c: complex SUCH THAT c.re > 0.0 :: zero
** c = zero),
        ALL(c x y: complex SUCH THAT c ~= zero
            :: c ** (x + y) = (c ** x) * (c ** y) )

MESSAGE equal(c1 c2: complex)
    REPLY(b: boolean)
    WHERE b <=> c1.re = c2.re & c1.im = c2.im
END
```

SAMPLE OUTPUT 12

```
TYPE complex
   INHERIT number{complex}

   MODEL (re im : real)
      INVARIANT true

   MESSAGE real_to_complex(r : real)
-- type conversion operation for mixed mode arithmetic

      REPLY (c : complex)
         WHERE c.re = r, c.im = 0.0

   MESSAGE rational_to_complex(r : rational)
-- type conversion operation for mixed mode arithmetic

      REPLY (c : complex)
         WHERE c = real_to_complex(rational_to_real@real(r))

   MESSAGE integer_to_complex(i : integer)
-- type conversion operation for mixed mode arithmetic

      REPLY (c : complex)
         WHERE c = real_to_complex(integer_to_real@real(i))

   MESSAGE nat_to_complex(n : nat)
-- type conversion operation for mixed mode arithmetic

      REPLY (c : complex)
         WHERE c = real_to_complex(nat_to_real@real(n))

   MESSAGE real(c : complex)
      REPLY (b : boolean)
         WHERE b <=> c.im = 0.0

   MESSAGE imaginary(c : complex)
      REPLY (b : boolean)
         WHERE b <=> c.re = 0.0

   MESSAGE rational(c : complex)
      REPLY (b : boolean)
         WHERE SOME(r : rational :: c = rational_to_complex(r))

   MESSAGE integral(c : complex)
      REPLY (b : boolean)
         WHERE SOME(i : integer :: c = integer_to_complex(i))

   MESSAGE nat(c : complex)
      REPLY (b : boolean)
      WHERE SOME(n : nat :: c = nat_to_complex(n))

MESSAGE zero
   REPLY (c : complex)
      WHERE c = real_to_complex(0.0)

MESSAGE one
   REPLY (c : complex)
      WHERE c = real_to_complex(1.0)

MESSAGE I
   REPLY (c : complex)
      WHERE I * I = -one

MESSAGE conjugate(c1 : complex)
   REPLY (c2 : complex)
      WHERE c2.re = c1.re, c2.im = -c1.im

MESSAGE magnitude(c1 : complex)
   REPLY (r : real)
      WHERE r = (c.re ** 2 + c.im ** 2) ** 0.5

MESSAGE minus(c1 : complex) OPERATOR -
   REPLY (c2 : complex)
      WHERE c2 = 0.0 - c1

MESSAGE plus(c1 c2 : complex) OPERATOR +
   REPLY (c3 : complex)
      WHERE c3.re = c1.re + c2.re, c3.im = c1.im + c2.im

MESSAGE difference(c1 c2 : complex) OPERATOR -
   REPLY (c3 : complex)
      WHERE c1 = c2 + c3

MESSAGE times(c1 c2 : complex) OPERATOR *
   REPLY (c3 : complex)
      WHERE c3.re = c1.re * c2.re - c1.im * c2.im,
            c3.im = c1.re * c2.im + c1.im * c2.re

MESSAGE quotient(c1 c2 : complex) OPERATOR /
   WHEN c2 ~= zero
      REPLY (c3 : complex)
         WHERE c1 = c2 * c3
   OTHERWISE
      REPLY EXCEPTION divide_by_zero

MESSAGE expt(c1 c2 : complex) OPERATOR **
   WHEN (c1 = zero & c2.re <= 0.0)
      REPLY EXCEPTION undefined_expt
   OTHERWISE
      REPLY (c3 : complex)
```

```
            WHERE ALL(c : complex :: c ** one = c),
                  ALL(c : complex SUCH THAT c.re > 0.0 :: zero ** c = zero),
                  ALL(c x y : complex SUCH THAT c ~= zero
                      :: c ** (x + y) = (c ** x) * (c ** y))

   MESSAGE equal(c1 c2 : complex)
      REPLY (b : boolean)
         WHERE b <=> c1.re = c2.re & c1.im = c2.im
END
```

SAMPLE INPUT 13

```
TYPE sequence{t: type}
INHERIT equality{sequence{t}}
INHERIT total_order{sequence{t}}
EXPORT sorted distinct permutation frequency
MODEL
-- generated by {empty, add}
INVARIANT true

MESSAGE empty
-- literal [ ] = empty
REPLY(s: sequence{t})

MESSAGE add(x: t, s1: sequence{t})
-- literal [x] = add(x, empty)
-- literal [x, $s] = add(x, s)
REPLY(s2: sequence{t})

MESSAGE remove(x: t, s1: sequence{t})
-- Remove all instances of x from s.
REPLY(s2: sequence{t})
WHERE ALL(x: t :: remove(x, empty) = empty),
ALL(x: t, s: sequence{t} :: remove(x, add(x, s)) = remove(x, s)),
ALL(x y: t, s: sequence{t} SUCH THAT x ~= y
:: remove(x, add(y, s)) = add(y, remove(x, s)) )

MESSAGE append(s1 s2: sequence{t}) OPERATOR ||
-- literal [$s1, $s2] = append(s1, s2)
REPLY(s2: sequence{t})
WHERE ALL(s: sequence{t} :: append(empty, s) = s),
ALL(x: t, s1 s2: sequence{t}
:: append(add(x, s1), s2) = add(x, append(s1, s2)) )

MESSAGE fetch(s: sequence{t}, n: nat) OPERATOR [
-- fetch(s, n) = s[n]
WHEN 1 <= n <= length(s)
REPLY(x: t)
WHERE ALL(x: t, s: sequence{t} :: fetch(add(x, s), 1) = x),
ALL(n: nat, x: t, s: sequence{t} SUCH THAT n > 1
:: fetch(add(x, s), n) = fetch(s, n - 1) )
OTHERWISE REPLY EXCEPTION bounds_error

MESSAGE fetch(s1: sequence{t}, s2: sequence{nat}) OPERATOR [
REPLY(s3: sequence{t})
WHERE length(s3) = length(s2),
ALL(n: nat SUCH THAT n IN domain(s2) :: s3[n] = s1[s2[n]])

MESSAGE length(s: sequence{t})
REPLY(n: nat)
```

```
WHERE length(empty) = 0,
ALL(x: t, s: sequence{t} :: length(add(x, s)) = length(s) + 1)

MESSAGE domain(s: sequence{t})
REPLY(d: set{nat})
WHERE d = {1 .. length(s)}

MESSAGE member(x: t, s: sequence{t}) OPERATOR IN
REPLY(b: boolean)
WHERE b <=> SOME(n: nat SUCH THAT n IN domain(s) :: s[n] = x)

MESSAGE equal(s1 s2: sequence{t})
REPLY(b: boolean)
WHERE b <=> ALL(n: nat :: s1[n] = s2[n])

MESSAGE less(s1 s2: sequence{t})
-- lexicographic ordering (dictionary ordering on strings)
WHEN has_operation(t, less) & partial_ordering(less@t)
REPLY(b: boolean)
WHERE ALL(s: sequence{t}, x: t :: [ ] < [x, $s]),
ALL(s1 s2: sequence{t}, x1 x2: t
:: [x1, $s1] < [x2, $s2] <=> x1 < x2 ¦ x1 = x2 & s1 < s2 )
OTHERWISE REPLY EXCEPTION operation_not_applicable

MESSAGE subsequence(s1 s2: sequence{t})
REPLY(b: boolean)
-- True if the elements of s1 are embedded in s2, in the same order.
WHERE ALL(s: sequence{t} :: subsequence([ ], s)),
ALL(s1 s2: sequence{t}, x: t :: subsequence([x, $s1], s2)
<=> SOME(s3 s4: sequence{t}
:: s2 = [$s3, x, $s4] & subsequence(s1, s4) ))

MESSAGE interval(x1 x2: t) OPERATOR ..
WHEN subtype(t, total_order)
REPLY(s: sequence{t})
WHERE sorted{less}(s) & ALL(x: t :: x IN s <=> x1 <= x <= x2)
OTHERWISE REPLY EXCEPTION operation_not_applicable

MESSAGE apply(f: FUNCTION, s1: sequence{t})
WHEN domain(f) = [t]
CHOOSE(rt: type SUCH THAT rt = range(f))
REPLY(s2: sequence{rt})
WHERE length(s2) = length(s1),
ALL(n: nat SUCH THAT n IN domain(s1) :: s2[n] = f(s1[n]))
OTHERWISE REPLY EXCEPTION type_error

MESSAGE reduce{f: FUNCTION}
WHERE domain(f) = [t, t] & range(f) = t
& SOME(x: t :: ALL(y: t :: f(y, x) = y))
(s: sequence{t})
REPLY(x: t)
```

```
WHERE IF s = [ ] THEN ALL(y: t :: f(y, x) = y)
ELSE x = f(s[1], reduce(s[2 .. length(s)], f)) FI

CONCEPT sorted{le: FUNCTION} WHERE total_ordering(le)
(s: sequence{t})
VALUE(b: boolean)
WHERE b <=> ALL(n1 n2: nat SUCH THAT 1 <= n1 < n2 <= length(s)
:: le(s[n1], s[n2]) )

CONCEPT distinct(s: sequence{t})
VALUE(b: boolean)
WHERE b <=> ALL(n1 n2: nat SUCH THAT 1 <= n1 < n2 <= length(s)
:: s[n1] ~= s[n2] )

CONCEPT permutation(s1 s2: sequence{t})
VALUE(b: boolean)
WHERE b <=> ALL(x: t :: frequency(x, s1) = frequency(x, s2))

CONCEPT frequency(x: t, s: sequence{t})
VALUE(n: nat)
WHERE n = NUMBER(k: nat SUCH THAT s[k] = x :: k)
END
```

SAMPLE OUTPUT 13

```
TYPE sequence{t : type}
   INHERIT equality{sequence{t}}
   INHERIT total_order{sequence{t}}
   EXPORT sorted distinct permutation frequency

   MODEL   -- generated by {empty, add}

      INVARIANT true

   MESSAGE empty  -- literal [ ] = empty

      REPLY (s : sequence{t})

   MESSAGE add(x : t, s1 : sequence{t})  -- literal [x] = add(x, empty)
-- literal [x, $s] = add(x, s)

      REPLY (s2 : sequence{t})

   MESSAGE remove(x : t, s1 : sequence{t})
-- Remove all instances of x from s.

      REPLY (s2 : sequence{t})
         WHERE ALL(x : t :: remove(x, empty) = empty),
               ALL(x : t, s : sequence{t}
                   :: remove(x, add(x, s)) = remove(x, s)),
               ALL(x y : t, s : sequence{t} SUCH THAT x ~= y
                   :: remove(x, add(y, s)) = add(y, remove(x, s)))

   MESSAGE append(s1 s2 : sequence{t}) OPERATOR ||
-- literal [$s1, $s2] = append(s1, s2)

      REPLY (s2 : sequence{t})
         WHERE ALL(s : sequence{t} :: append(empty, s) = s),
               ALL(x : t, s1 s2 : sequence{t}
                   :: append(add(x, s1), s2) = add(x, append(s1, s2)))

   MESSAGE fetch(s : sequence{t}, n : nat) OPERATOR [   -- fetch(s, n) = s[n]

      WHEN 1 <= n <= length(s)
         REPLY (x : t)
            WHERE ALL(x : t, s : sequence{t} :: fetch(add(x, s), 1) = x),
                  ALL(n : nat, x : t, s : sequence{t} SUCH THAT n > 1
                      :: fetch(add(x, s), n) = fetch(s, n - 1))
      OTHERWISE
         REPLY EXCEPTION bounds_error

   MESSAGE fetch(s1 : sequence{t}, s2 : sequence{nat}) OPERATOR [
      REPLY (s3 : sequence{t})
      WHERE length(s3) = length(s2),
            ALL(n : nat SUCH THAT n IN domain(s2) :: s3[n] = s1[s2[n]])

  MESSAGE length(s : sequence{t})
    REPLY (n : nat)
      WHERE length(empty) = 0,
            ALL(x : t, s : sequence{t} :: length(add(x, s)) = length(s) + 1)

  MESSAGE domain(s : sequence{t})
    REPLY (d : set{nat})
      WHERE d = {1 .. length(s)}

  MESSAGE member(x : t, s : sequence{t}) OPERATOR IN
    REPLY (b : boolean)
      WHERE b <=> SOME(n : nat SUCH THAT n IN domain(s) :: s[n] = x)

  MESSAGE equal(s1 s2 : sequence{t})
    REPLY (b : boolean)
      WHERE b <=> ALL(n : nat :: s1[n] = s2[n])

  MESSAGE less(s1 s2 : sequence{t})
-- lexicographic ordering (dictionary ordering on strings)

    WHEN has_operation(t, less) & partial_ordering(less@t)
      REPLY (b : boolean)
        WHERE ALL(s : sequence{t}, x : t :: [] < [x, $s]),
              ALL(s1 s2 : sequence{t}, x1 x2 : t
                 :: [x1, $s1] < [x2, $s2] <=> x1 < x2 ¦ x1 = x2 & s1 < s2)
    OTHERWISE
      REPLY EXCEPTION operation_not_applicable

  MESSAGE subsequence(s1 s2 : sequence{t})
    REPLY (b : boolean)
-- True if the elements of s1 are embedded in s2, in the same order.

      WHERE ALL(s : sequence{t} :: subsequence([], s)),
            ALL(s1 s2 : sequence{t}, x : t
               :: subsequence([x, $s1], s2)
                  <=> SOME(s3 s4 : sequence{t}
                        :: s2 = [$s3, x, $s4] & subsequence(s1, s4)))

  MESSAGE interval(x1 x2 : t) OPERATOR ..
    WHEN subtype(t, total_order)
      REPLY (s : sequence{t})
        WHERE sorted{less}(s) & ALL(x : t :: x IN s <=> x1 <= x <= x2)
    OTHERWISE
      REPLY EXCEPTION operation_not_applicable

  MESSAGE apply(f : FUNCTION, s1 : sequence{t})
    WHEN domain(f) = [t]
      CHOOSE(rt : type SUCH THAT rt = range(f))
```

```
        REPLY (s2 : sequence{rt})
           WHERE length(s2) = length(s1),
                 ALL(n : nat SUCH THAT n IN domain(s1) :: s2[n] = f(s1[n]))
      OTHERWISE
        REPLY EXCEPTION type_error

   MESSAGE reduce{f : FUNCTION}
      WHERE domain(f) = [t, t] & range(f) = t
            & SOME(x : t :: ALL(y : t :: f(y, x) = y))
      (s : sequence{t})
      REPLY (x : t)
         WHERE IF s = []
               THEN ALL(y : t :: f(y, x) = y)
               ELSE x = f(s[1], reduce(s[2 .. length(s)], f)) FI

   CONCEPT sorted{le : FUNCTION}
      WHERE total_ordering(le)(s : sequence{t})
      VALUE (b : boolean)
      WHERE b
            <=> ALL(n1 n2 : nat SUCH THAT 1 <= n1 < n2 <= length(s)
                    :: le(s[n1], s[n2]))

   CONCEPT distinct(s : sequence{t})
      VALUE (b : boolean)
      WHERE b
            <=> ALL(n1 n2 : nat SUCH THAT 1 <= n1 < n2 <= length(s)
                    :: s[n1] ~= s[n2])

   CONCEPT permutation(s1 s2 : sequence{t})
      VALUE (b : boolean)
      WHERE b <=> ALL(x : t :: frequency(x, s1) = frequency(x, s2))

   CONCEPT frequency(x : t, s : sequence{t})
      VALUE (n : nat)
      WHERE n = NUMBER(k : nat SUCH THAT s[k] = x :: k)
END
```

SAMPLE INPUT 14

```
MACHINE airline_reservation_system
STATE(fl: map{flight_id, flight}, res: set{reservation})
INVARIANT ALL(r: reservation :: r IN res => r.id IN fl)
INITIALLY domain(fl) = { }, res = { }

-- interface to the travel_agent
MESSAGE get_flight_info
    (from to: city, earliest_departure lastest_arrival: time)
REPLY(s: set{flight})
WHERE ALL(f: flight
                  :: f IN s <=> f.origin = from & f.destination = to &
                f.departure >= earliest_departure & f.arrival <= latest_arrival )

MESSAGE reserve(name: passenger, id: flight_id, day: date)
WHEN id IN fl & number_res(id, day, res) < fl[id].capacity & ~has_res(name, id, day, res)
REPLY(s: set{seat_id})
WHERE ALL(si: seat_id :: si IN s <=> unassigned(si, id, day, res) )
TRANSITION SOME(r: reservation
                SUCH THAT r.who = name & r.id = id & r.when = day & r.seat = !
                :: res = union(*res, {r}))
WHEN id IN fl & has_res(name, id, day)
REPLY EXCEPTION previous_reservation
WHEN ~(id IN fl)
REPLY EXCEPTION invalid_flight_id
OTHERWISE REPLY EXCEPTION no_seats_available
MESSAGE cancel_reservation(id: flight_id, day: date, name: passenger)
WHEN id IN fl & has_res(name, id, day, res)
REPLY(confirmation: string)
WHERE confirmation = "reservation cancelled"
TRANSITION res = *res - {find_res(id, day, name, res)}
WHEN ~(id IN fl)
REPLY EXCEPTION invalid_flight_id
OTHERWISE REPLY EXCEPTION no_such_reservation

MESSAGE assign_seat
    (id: flight_id, day: date, name: passenger, seat: seat_id)
WHEN has_res(name, id, day, res) & unassigned(seat, id, day, res)
REPLY(confirmation: string)
TRANSITION find_res(id, day, name, res).seat = seat
WHEN has_res(name, id, day, res) & ~unassigned(seat, id, day, res)
REPLY EXCEPTION seat_not_available
OTHERWISE REPLY EXCEPTION no_such_reservation

-- interface to the airline_manager
MESSAGE update_price(id: flight_id, price: money)
WHEN id IN fl
REPLY(confirmation: string)
TRANSITION fl[id].price = price
```

```
OTHERWISE REPLY EXCEPTION no_such_flight

MESSAGE add_flight(id: flight_id, origin destination: airport,
                      departure arrival: time, capacity: integer, price: money)
WHEN ~(id IN fl)
REPLY(confirmation: string)
WHERE confirmation = "flight added"
TRANSITION fl[id] = create@flight(id, origin, destination, departure,
                                          arrival, capacity, price)
OTHERWISE REPLY EXCEPTION flight_already_exists

MESSAGE drop_flight(id: flight_id)
WHEN id IN fl
REPLY(confirmation: string)
WHERE confirmation = "flight dropped"
TRANSITION fl[id] = !
OTHERWISE REPLY EXCEPTION no_such_flight

-- concepts
CONCEPT number_res(id: flight_id, d: date, rs: set{reservation})
VALUE(n: integer)
    -- the number of reservations held for flight id on day d
WHERE SOME(s: set{reservation} SUCH THAT
            ALL(r: reservation :: r IN s <=> r IN rs & r.when = day & r.flight = id )
          :: n = size(s) )

CONCEPT has_res(name: passenger, id: flight_id, day: date, rs: set{reservation})
VALUE(b: boolean)
    -- true if the passenger holds a reservation on day for flight id
WHERE b <=> SOME(r: reservation
                 SUCH THAT r.who = name & r.flight = id & r.when = day
                 :: r IN rs )
CONCEPT unassigned(si: seat_id, id: flight_id, day: date, rs: set{reservation})
VALUE(b: boolean)
  -- true if no one holds a reservation for seat si on day for id
 WHERE b <=> ALL(r: reservation
                 SUCH THAT r.seat = si & r.flight = id & r.when = day
                 :: ~(r IN rs) )
CONCEPT reservation: type
 WHERE reservation = tuple{who :: passenger, when :: date, id :: flight_id,
                              seat :: seat_id }
CONCEPT city: type
CONCEPT flight_id: type WHERE flight_id = string
CONCEPT passenger: type     WHERE passenger = string
CONCEPT date: type    CONCEPT seat_id: type     WHERE seat_id = string
CONCEPT money: type    CONCEPT airport: type     WHERE airport = string
END  TYPE flight  MODEL(?)   INVARIANT ? END
TYPE time   INHERIT total_order   MODEL(?)   INVARIANT ? END
```

SAMPLE OUTPUT 14

```
MACHINE airline_reservation_system

   STATE (fl : map{flight_id, flight}, res : set{reservation})
      INVARIANT ALL(r : reservation :: r IN res => r.id IN fl)
      INITIALLY domain(fl) = {}, res = {}  -- interface to the travel_agent


   MESSAGE get_flight_info(from to : city, earliest_departure lastest_arrival
                           : time)
      REPLY (s : set{flight})
         WHERE ALL(f : flight
                  :: f IN s
                     <=> f.origin = from & f.destination = to
                        & f.departure >= earliest_departure
                        & f.arrival <= latest_arrival)

   MESSAGE reserve(name : passenger, id : flight_id, day : date)
      WHEN id IN fl & number_res(id, day, res) < fl[id].capacity
           & ~has_res(name, id, day, res)
         REPLY (s : set{seat_id})
            WHERE ALL(si : seat_id :: si IN s <=> unassigned(si, id, day, res))
         TRANSITION SOME(r : reservation
                          SUCH THAT r.who = name & r.id = id & r.when = day
                                 & r.seat = | :: res = union(*res, {r}))
      WHEN id IN fl & has_res(name, id, day)
         REPLY EXCEPTION previous_reservation
      WHEN ~(id IN fl)
         REPLY EXCEPTION invalid_flight_id
      OTHERWISE
         REPLY EXCEPTION no_seats_available

   MESSAGE cancel_reservation(id : flight_id, day : date, name : passenger)
      WHEN id IN fl & has_res(name, id, day, res)
         REPLY (confirmation : string)
            WHERE confirmation = "reservation cancelled"
         TRANSITION res = *res - {find_res(id, day, name, res)}
      WHEN ~(id IN fl)
         REPLY EXCEPTION invalid_flight_id
      OTHERWISE
         REPLY EXCEPTION no_such_reservation

   MESSAGE assign_seat(id : flight_id, day : date, name : passenger, seat
                       : seat_id)
      WHEN has_res(name, id, day, res) & unassigned(seat, id, day, res)
         REPLY (confirmation : string)
         TRANSITION find_res(id, day, name, res).seat = seat
      WHEN has_res(name, id, day, res) & ~unassigned(seat, id, day, res)
         REPLY EXCEPTION seat_not_available
      OTHERWISE
         REPLY EXCEPTION no_such_reservation
-- interface to the airline_manager


   MESSAGE update_price(id : flight_id, price : money)
      WHEN id IN fl
         REPLY (confirmation : string)
         TRANSITION fl[id].price = price
      OTHERWISE
         REPLY EXCEPTION no_such_flight

   MESSAGE add_flight(id : flight_id, origin destination : airport, departure
                      arrival : time, capacity : integer, price : money)
      WHEN ~(id IN fl)
         REPLY (confirmation : string)
            WHERE confirmation = "flight added"
         TRANSITION fl[id]
                    = create@flight(id, origin, destination, departure, arrival,
                                    capacity, price)
      OTHERWISE
         REPLY EXCEPTION flight_already_exists

   MESSAGE drop_flight(id : flight_id)
      WHEN id IN fl
         REPLY (confirmation : string)
            WHERE confirmation = "flight dropped"
         TRANSITION fl[id] = !
      OTHERWISE
         REPLY EXCEPTION no_such_flight  -- concepts


   CONCEPT number_res(id : flight_id, d : date, rs : set{reservation})
      VALUE (n : integer)
-- the number of reservations held for flight id on day d

      WHERE SOME(s : set{reservation}
                SUCH THAT ALL(r : reservation
                              :: r IN s
                                 <=> r IN rs & r.when = day & r.flight = id)
                :: n = size(s))

   CONCEPT has_res(name : passenger, id : flight_id, day : date, rs
                  : set{reservation})
      VALUE (b : boolean)
-- true if the passenger holds a reservation on day for flight id

      WHERE b
            <=> SOME(r : reservation
                     SUCH THAT r.who = name & r.flight = id & r.when = day
                     :: r IN rs)
   CONCEPT unassigned(si : seat_id, id : flight_id, day : date, rs
                     : set{reservation})
      VALUE (b : boolean)
-- true if no one holds a reservation for seat si on day for id

      WHERE b
          <=> ALL(r : reservation
                  SUCH THAT r.seat = si & r.flight = id & r.when = day
                  :: ~(r IN rs))

   CONCEPT reservation: type
      WHERE reservation
          = tuple{who :: passenger, when :: date, id :: flight_id, seat ::
                  seat_id}

   CONCEPT city: type

   CONCEPT flight_id: type
      WHERE flight_id = string

   CONCEPT passenger: type
      WHERE passenger = string

   CONCEPT date: type

   CONCEPT seat_id: type
      WHERE seat_id = string

   CONCEPT money: type

   CONCEPT airport: type
      WHERE airport = string
END

TYPE flight

   MODEL (?)
      INVARIANT ?
END

TYPE time
   INHERIT total_order

   MODEL (?)
      INVARIANT ?
END
```

## LIST OF REFERENCES

1. Berzins, Valdis, unpublished manuscript, Naval Postgraduate School, Department of Computer Science, Monterey, California.

2. Grammar for SPEC provided by thesis advisor, Valdis Berzins, Naval Postgraduate School, Department of Computer Science, Monterey, California.

3. Sample input provided by thesis advisor, Valdis Berzins, Naval Postgraduate School, Department of Computer Science, Monterey, California.

4. Lu, P.M., Yau, S.S., and Hong, W., "A Formal Methodology Using Attributed Grammars for Multiprocessing-System Software Development. I. Design Representation", Information Sciences, v. 30, pp. 79-105, August 1983.

5. Hopcroft, John E. and Ullman, Jeffrey D., Introduction to Automata Theory, Languages, and Computation, Addison-Wesley Publishing Company, Inc., 1979.

6. Lewis, Harry R. and Papadimitriou, Christos H., Elements of the Theory of Computation, Prentice-Hall, Inc., 1981.

7. Boccalatte, A., Di Manzo, M., and Sciatta, D., "Error Recovery with Attribute Grammars", Computer Journal, v. 25, pp. 331-337, August 1982.

8. Knuth, Donald E., "Semantics of Context Free Languages", Mathematical Systems Theory, v. 2, pp. 127-145, 1968.

9. Institute of Technology, University of Minnesota, Computer Science Department Report TR 85-37, The Incompleat AG Guide and Reference Manual, by Robert M. Herndon, Jr, October 1985.

10. Rubin, L.F., "Syntax-Directed Pretty Printing - A First Step Towards a Syntax-Directed Editors", IEEE Transactions on Software Engineering, v. SE-9, pp. 119-127, March 1983.

11. Documentation code from file entitled KCLIB.C, provided by thesis advisor, Valdis Berzins, Naval Postgraduate School, Department of Computer Science, Monterey, California.

# BIBLIOGRAPHY

Adorni, G., Boccolatte, A., and Di Manzo, M., "Top-Down Semantic Analysis", Computer Journal, v. 27, August 1984.

Ahn, Tae Nam, Program Family for Extended Pretty Printer, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1983.

Bochman, Gregor V., "Semantic Evaluation from Left to Right", Communications of the ACM, v. 19, 2 February 1976.

Chebotar', K.S., "Some Modifications of Knuth's Algorithm for Verifying Cyclicity of Attribute Grammars", Programming and Computer Software, v. 7, January-February 1981.

Courcelle, B., "Attribute Grammars: Theory and Applications", Lecture Notes in Computer Science, v. 107, 19-25 April 1981.

Courcelle, Bruno and Franchi-Zanettacci, Paul, "On the Expressive Power of Attribute Grammars", Proceedings of the 21st Annual Symposium on Foundations of Computer Science, 1980.

Courcelle, Bruno and Franchi-Zanettacci, Paul, "Attribute Grammars and Recursive Program Schemes", Theoretical Computer Science, v. 17, 1982.

Dembinski, Piotr and Maluszynski, Jan, "Attribute Grammars and Two-Level Grammars: A Unifying Approach", Mathematical Foundations of Computer Science, 4-8 September 1978.

Deransart, Pierre, "Logical Attribute Grammars", Information Processing, v. 9, 19-23 September 1983.

Dowling, William F., "Attribute Grammar Interpretation as a Model of Computation", ACM Fourteenth Annual Computer Science Conference: CSC '86 Proceedings, 4-6 February 1986.

Farrow, Rodney, "Generating a Production Compiler from an Attribute Grammar", IEEE Software, v. 1, October 1984.

Felice, Robert, "Pretty Printer", Commodore, v. 5, July/August 1984.

Fisher, Robert C., "A Practical Exercise in Program Design", Journal of Pascal and Ada (USA), v. 2, September/October 1983.

Institute of Technology, University of Minnesota, Computer Science Department Report TR 85-25, AG: A Useful Attribute Grammar Translator Generator, by Robert M. Herdon, Jr and Valdis A. Berzins, July 1985.

Jia, Xiaoping and Qian, Jiahua, "Incremental Evaluation of Attributed Grammars for Incremental Programming Environments", Proceedings of COMPSAC 85, 9-11 October 1985.

Kamimura, Tsutomo, "Tree Automata and Attribute Grammars", Information and Control, v. 57, April 1983.

Mahoy, Brian H., "Attribute Grammars and Mathematical Semantics", Society for Industrial and Applied Mathematics Journal Computers, v. 10, August 1981.

Massachusetts Institute of Technology, Cambridge. Artificial Intelligence Lab Report AI-M-611, Gprint: A LISP Pretty Printer Providing Extensive User Format-Control Mechanism, by Richard C. Waters, October 1981.

Massachusetts Institute of Technology, Cambridge. Artificial Intelligence Lab Report AI-M-816, PP: A LISP Pretty Printing System, by Richard C. Waters, December 1984.

Oppen, Derek C., "Prettyprinting", ACM Transactions on Programming Languages and Systems, v. 2, October 1980.

Raiha, Kari-Jouko, "Bibliography on Attribute Grammars", ACM Sigplan Notices, v. 15, 1980.

Tartan Labs Report TL-83-3, A Diana-Driven Pretty-Printer for ADA, by Kenneth J. Butler and Arthur Evans Jr, 22 February 1983.

Tartan Labs Report TL-83-36, Extensions to Attribute Grammars, by J.R. Nestor, B. Mishra, W.L. Scherlis, and W.A. Wulf, April 1983.

## INITIAL DISTRIBUTION LIST

| | | No. Copies |
|---|---|---|
| 1. | Defense Technical Information Center<br>Cameron Station<br>Alexandria, Virginia 22304-6145 | 2 |
| 2. | Library, Code 0142<br>Naval Postgraduate School<br>Monterey, California 93943-5002 | 2 |
| 3. | Director, Information Systems (OP-945)<br>Office of the Chief of Naval Operations<br>Navy Department<br>Washington, D. C. 20350-2000 | 1 |
| 4. | Superintendent, Naval Postgraduate School<br>Computer Technology Programs, Code 37<br>Monterey, California  93943-5000 | 1 |
| 5. | Robert H. Herndon, Jr.<br>Institute of Technology<br>University of Minnesota<br>136 Lind Hall/207 Church Street S.E.<br>Minneapolis, Minnesota 55455 | 1 |
| 6. | Professor Valdis Berzins, Code 52Be<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, California 94943-5000 | 1 |
| 7. | Professor Lucia Luqi, Code 52Lq<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 1 |
| 8. | LT Jill A. Weigand<br>c/o Mr. Richard Forsythe<br>127 Brass Road<br>Box 154<br>Lakeside, Montana 59922 | 2 |

21 - 237