```
/*****************************************************************************************************
    This grammar is intended more for readability rather than formal precision, non-ambiguity, etc.  We
use the following conventions.
    A + at the end of any subproduction in parentheses means 1 or more.
    A ? at the end of any subproduction in parentheses means 0 or 1.

    Any non-terminal ending with "List" is a list of one or more substrings, each matching the
corresponding rule.  Elements in such a list may need a separator, like comma, semicolon etc, (if so,
this is mentioned in comments).  For example:

  ExpressionList           is a list of one or more expressions.
  (ExpressionList)?        is a list of ZERO or more expressions

  Identifiers are used in Alphabets in three ways: names of external functions, names of index-variables
(and program size parameters), and names of data-variables,  Except in a very specific situation
described below the two cannot be interchanged -- index expressions are affine expressions that use only
index variables and parameters.  Data-variables are used to construct (data) expressions.  The grammar is
written in a "top-down" manner, and the rules for new nonterminals are presented in a depth-first order.

*****************************************************************************************************/

/*  A program consists of 0 or more ExternalFunctionPrototypes followed by 1 or more AlphabetsSystems. */
AlphabetsProgram ::= (ExternalFunctionPrototypeList) AlphabetsSystemList

/* External function prototypes (signatures) may have zero or more, comma-separated, formal parameters. */
ExternalFunctionPrototype ::= Type ID "(" (TypeList)? ")" ";" /* TypeList is comma-separated */

Type ::= "int" |"float" |"double" |"bool" |"longlong" |"long" |"short" |"char"

AlphabetsSystem ::= "affine" ID                    /* ID is the system name. */
                    (Domain)?                      /* Size parameters */
                    "given" (VariableDeclList)?    /* Input variable declarations, comma separated list */
                    "returns" (VariableDeclList)?  /* Output variable declarations */
                    ("using" (VariableDeclList)?)? /* Local variable declarations */
                    "through" (EquationList)?      /* declarations/equations are optional */
                    "."

Domain ::= Domain "||" Domain        /* The logical union of domains */
           | "{" (IDList)? "|"       /* IDList is comma-separated */
                InEquationList "}"   /* InEquationList separator is && */

InEquation ::= IndexExpList  /* IndexExpList separator is a ComparisonOp. All ComparisonOps used
                             in a single inequation must be "transitively compatible,"
                             i.e., a<=b>=c is not acceptable but a<=b<c is, as is a>b==c */

IndexExp ::= Integer       /* special syntax for affine functions with constant coefficients */
           | ID
           | Integer ID
           | IndexExp ( "+" | "-" ) IndexExp

VariableDecl ::= Type IDList (Domain)? ";" /* Scalar variables do not need a domain. */

Equation ::= ID "=" Expression ";"                   /* i.e., (data) expression */
           | ID "[" (IDList)? "]" "=" Expression ";" /* zero or more, comma-separated IDs on lhs */

Expression ::= ID            /* (Data) Variable. */
             | Constant      /* an integer, char, float, double, bool, etc. literal */
             | PointwiseOpExpression
             | DependenceExpression
             | CaseExpression
             | RestrictExpression
             | "[" IndexExpression "]"  /* coersion of an index expression to a (data) expression */
             | ReduceExpression
             | "(" Expression ")"  /* parenthesized to override default precedence */

PointwiseOpExpression ::= IfThenElseExpression
                        | ExternalFunctionExpression
```

```
                        | BuiltInOpExpression

IfThenElseExpression ::= "if"  Expression  /* type bool */
                         "then" Expression
                         "else" Expression /* The three subexpressions must be type-compatible */

ExternalFunctionExpression ::= ID "(" (ExpressionList)? ")" /* comma-separated, possibly empty */

BuiltInOpExpression ::= Expression Op Expression         /* Operators have the standard precedence. */
                      | "min" "(" Expression "," Expression ")"
                      | "max" "(" Expression "," Expression ")"

Op ::= ArithmeticOp | LogicOp | ComparisonOp

ArithmeticOp ::= "+" | "-" | "*" | "/"

LogicOp ::= "&&" | "||"

ComparisonOp ::= ">" | "<" | ">=" | "<=" | "=="

/* There are two kinds of dependence expressions, but both cannot be arbitrarily nested or intermingled
in the same expression.  The second form is more general but the first one is more intuitive and usually
suffices. */

DependenceExpression ::= ID "[" IndexExpressionList "]"  /* comma-separated list of index expressions */
                       | Dep "@" Expression

Dep ::= "(" (IDList)? "->" (IndexExpressionList)? ")" /* comma-separated, possibly empty lists */

CaseExpression ::= "case" ExpressionList "esac"

RestrictedExpression ::= Domain ":" Expression

/* Reduction expressions can be written in two styles. */
ReduceExp ::= "reduce" "(" ReductionOp "," Projection "," Expression ")"
            | "MAX" "(" Projection "," Expression ")"
            | "MIN" "(" Projection "," Expression ")"
            | "SUM" "(" Projection "," Expression ")"

ReductionOp::="+"|"*"|"max"|"min"|"and"|"or"|"xor"

Projection ::= "[" (IDList)? "]" /* comma-separated, may be empty */
             | Dep
```