Alloy 3.0 Reference Manual

Daniel Jackson
May 10, 2004

Contributors to the design of the Alloy language include: Daniel Jackson, Manu Sridharan, Ilya Shlyakhter, Emina Torlak, Jonathan Edwards, Greg Dennis, Edmond Lau, Robert Seater, Derek Rayside, Mandana Vaziri, Sarfraz Khurshid, Mana Taghdiri, Ian Schechter, Uriel Schafer, and Joseph Cohen.

Contents

1 Introduction

Alloy is a lightweight modelling language for software design. It is amenable to a fully automatic analysis, using the Alloy Analyzer. Information about the Alloy project is available on its web page, http: alloy.mit.edu.

This manual summarizes the language. It is not likely to be suitable as a tutorial. An online tutorial is available on the website, and a book about modelling with Alloy is forthcoming.

2 Lexical Issues

The permitted characters are the printing characters of the ASCII character set, with the exception of:

- backslash \
- backquote `

and, of the ASCII non-printing characters, only space, horizontal tab, carriage return and linefeed. Since the encoding of linebreaks varies across platforms, the Alloy Analyzer accepts any of the standard combinations of carriage and linefeed.

The non-alphanumeric symbols are used as operators or for punctuation, with the exception of

- dollar sign $;
- percent sign %;
- question mark ?;
- underscore _;
- single and double quote marks (‘ and “).

Dollar, percent are question mark are reserved for use in future versions of the language. Underscore and quotes may be used in identifiers. Single and double quote marks (numbered 39 and 34 in ASCII) should not be confused with typographic quote marks and the prime mark, which are not acceptable characters. If text is prepared in a word processor, ensure that a ‘smart quotes’ feature is not active, since it might generate typographic quote marks from simple ones.

Characters between -- or // and the end of the line, and from /* to */, are treated as comments. Multiple-line comments may not be nested.

Non-comment text is broken into tokens by the following separators:

- whitespace (space, tab, linebreak);
- non-alphanumeric characters (except for underscore and quote marks).

The meaning of the text is independent of its format; in particular, linebreaks are treated as whitespace just like spaces and tabs.

Keywords and identifiers are case sensitive.

Identifiers may include any of the alphabetic characters, and, except as the first character, numbers, underscores, question mark and exclamation point, and quote marks. A hyphen may not appear in an identifier, since it is treated as an operator.

A numeric constant consists of a sequence of digits between 0 and 9, whose first digit is not zero.

The following sequences of characters are recognized as single tokens:

- the double colon :: used for receiver syntax;
- the implication operator =>;
- the integer comparison operators >= and =<;
- the product arrow ->;
- the restriction operators <: and :>;
- the relational override operator ++;
- conjunction && and disjunction ||;
- the comment markings --, //, /* and */.

The negated operators (such as !=) are not treated as single tokens, so they may be written with whitespace between the negation and comparison operators.

The following are reserved as keywords and may not be used for identifiers:

abstract	all	and	as	assert
but	check	disj	else	exactly
extends	fact	for	fun	iden
if	iff	implies	in	Int

int	let	lone	module	no
none	not	one	open	or
part	pred	run	set	sig
some	sum	then	univ	

3 Namespaces

Each identifier belongs to a single namespace. There are three namespaces:

- Module names and module aliases;
- Signatures, fields, paragraphs (facts, functions, predicates and assertions), and bound variables (arguments to functions and predicates, and variables bound by let and quantifiers);
- Command names.

Identifiers in different namespaces may share names without risk of name conflict. Within a namespace, the same name may not be used for different identifiers with one exception: bound variables may shadow each other, and may shadow field names. Conventional lexical scoping applies; the innermost binding applies.

4 Grammar

The grammar uses the standard BNF operators:

- x* for zero or more repetitions of x;
- x⁺ for one or more repetitions of x;
- x | y for a choice of x or y;
- *[*x*]* for an optional x.

In addition,

- x,* means zero or more comma-separated occurrences of x;
- x,⁺ means one or more comma-separated occurrences of x.

To avoid confusion with grammar symbols, square brackets, star, plus and the vertical bar are set in bold type when they are to be interpreted as terminals.

Every name ending Id is an identifier, and is to be treated as a terminal. The terminal number represents a numeric constant.

module ::= header import* paragraph*
header ::= **module** *[*path*]* moduleId *[***[** sigId,⁺ **]***]*

path ::= directoryId / *[*path*]*
import ::= **open** *[*path*]* moduleId *[***[** sigRef,* **]***]* *[***as** aliasId*]*

paragraph ::=
  sigDecl | factDecl | funDecl | predDecl | assertDecl | runCmd |
checkCmd

sigDecl ::=
  *[***abstract***]* *[*mult*]* **sig** sigID,+ *[***extends** sigRef*]* sigBody
  | *[*mult*]* **sig** sigID,+ **in** sigRef sigBody
sigBody ::= **{** decl,* **}** *[*formulaSeq*]*

factDecl ::= **fact** *[*factId*]* formulaSeq
assertDecl ::= **assert** *[*assertId*]* formulaSeq
funDecl ::= **fun** *[*sigRef **::***]* funId **(** decl,***) :** declExpr **{** expr **}**
predDecl ::= **pred** *[*sigRef **::***]* predId **(** decl,* **)** formulaSeq

runCmd ::=
  *[*commandId **:***]* **run** funRef *[*scope*]*
  *[*commandId **:***]* **run** predRef *[*scope*]*
checkCmd ::= *[*commandId **:***]* **check** assertRef *[*scope*]*

scope ::= **for** number
  | **for** *[*number **but***]* typescope,+
typescope ::= *[***exactly***]* number scopeable
scopeable ::= sigRef | **int**

decl ::= *[***part** | **disj***]* varId,+ **:** declExpr
letDecl ::= varId **=** expr
declExpr ::=
  *[*mult | **set***]* expr
  | expr *[*mult*]* **->** *[*mult*]* expr
mult ::= **lone** | **one** | **some**

expr ::= *[***@***]* varId | sigRef | **this** |
  | **none** | **univ** | **iden**
  | unOp expr | expr binOp expr | expr**[** expr **]**
  | **{** decl,+ **|** *[*formula*]* **}**
  | **let** letDecl,+ **|** expr
  | **if** formula **then** expr **else** expr

```
    | Int intExpr
    | [expr ::] funRef ( expr,* )
    | ( expr )

intExpr ::= number | # expr | sum expr | int expr
    | if formula then intExpr else intExpr
    | intExpr intOp intExpr
    | let letDecl,... | intExpr
    | sum decl,+ | intExpr
    | ( intExpr )
intOp ::= + | -

formulaBody ::= formulaSeq | | formula
formulaSeq ::= { formula* }
formula ::= expr [neg] compOp expr
    | quantifier expr
    | intExpr [neg] intCompOp intExpr
    | neg formula | formula logicOp formula
    | formula thenOp formula [elseOp formula]
    | quantifier decl,+ formulaBody
    | let letDecl,+ formulaBody
    | [expr ::] predRef ( expr,* )
    | expr : declExpr
    | formulaSeq
    | ( formula )

thenOp ::= implies | =>
elseOp ::= else | ,

neg ::= not | !
logicOp ::= && | || | iff | <=> | and | or
quantifier ::= all | no | mult
binOp ::= + | - | & | . | -> | <: | :> | ++
unOp ::= ~ | * | ^
compOp ::= in | =
```

intCompOp ::= < | > | = | =< | >= funRef ::= [moduleRef] funId predRef ::= [moduleRef] predId assertRef ::= [moduleRef] assertId sigRef ::= [moduleRef] sigId | Int | univ moduleRef ::= [path] moduleId [[sigRef,*]] / | aliasId /

5 Precedence and Associativity

The precedence order for logical operators, tightest first, is:

- negation operators: ! and not;
- conjunction: && and and;
- implication: =>, <=>, implies and iff;
- disjunction: || and or.

The precedence order for expression operators, tightest first, is:

- unary operators: ~, ^ and *;
- restriction operators: <: and :>;
- dot join: . ;
- square brackets join: [];
- arrow product: ->;
- intersection: &;
- override: ++;
- union and difference: + and -.

Note that in particular dot binds more tightly than square brackets, so a.b[c] is parsed as (a.b)[c].

All binary operators are associative, except for: the logical implication operator, and the expression operators dot, intersection and difference. Implication associates to the right, and the expression operators associate to the left. So, for example, p => q => r is parsed as p => (q => r), and a.b.c is parsed as (a.b).c.

In an implication, an else-clause is associated with its closest then-clause. So the formula

```
p => q => r, s
```

for example, is parsed as

```
p => (q => r, s)
```

6 Semantic Basis

6.1 Instances and Meaning

A model's meaning is several collections of *instances*. An instance is a binding of values to variables. Typically, a single instance represents a state, or a pair of states (corresponding to execution of an

operation), or an execution trace. The language has no built-in notion of state machines, however, so an instance need not represent any of these things.

The collections of instances assigned to a model are:

- A set of *core instances* associated with the facts of the model, and the constraints implicit in the signature declarations. These instances have as their variables the signatures and their fields, and they bind values to them that make the facts and declaration constraints true.
- For each function or predicate, a set of those instances for which the facts and declaration constraints of the model as a whole are true, and additionally the constraint of the function or predicate are true. The variables of these instances are those of the core instances, extended with the arguments of the function or predicate.
- For each assertion, a set of those instances for which the facts and declaration constraints of the model as a whole are true, but for which the constraint of the assertion is false.

A model without any core instances is *inconsistent*, and almost certainly erroneous. A function or predicate without instances is likewise inconsistent, and is unlikely to be useful. An assertion is expected not to have any instances: the instances are *counterexamples*, which indicate that the assertion does not follow from the facts.

The Alloy Analyzer finds instances of a model automatically by search within finite bounds (specified by the user as a *scope*; see Section 8.5 below). Because the search is bounded, failure to find an instance does not necessarily mean that one does not exist. But instances that are found are guaranteed to be valid.

6.2 Relational Logic

Alloy is a first order relational logic. The values assigned to variables, and the values of expressions evaluated in the context of a given instance, are *relations*. These relations are first order: that is, they consist of tuples whose elements are atoms (and not themselves relations).

Alloy has no explicit notion of sets, scalars or tuples. A set is simply a unary relation; a scalar is a singleton, unary relation; and a tuple

is a singleton relation. The type system distinguishes sets from relations because they have different arity, but does not distinguish tuples and scalars from more general relations.

There is no function application operator. Relational join is used in its place, and is syntactically cleaner that it would be in a language that distinguished sets and scalars. For example, given a relation f that is functional, and x and y constrained to be scalars, the formula

```
x.f = y
```

constrains the image of x under the relation f to be the set y. So long as x is in the domain of f, this formula will have the same meaning as it would if the dot were interpreted as function application, f as a function, and x and y as scalar-typed variables. But if x is out of the domain of f, the expression x.f will evaluate to the empty set, and since y is constrained to be a scalar (that is, a singleton set), the formula as a whole will be false. In a language with function application, various meanings are possible, depending on how partial functions are handled. An advantage of the Alloy approach is that it sidesteps this issue.

The declaration syntax of Alloy has been designed so that familiar forms have their expected meaning. Thus, when X is a set, the quantified formula

```
all x: X | F
```

has x range over scalar values. That is, the formula F is evaluated for bindings of x to singleton subsets of X.

The syntax of Alloy does in fact admit higher-order quantifications. For example, the assertion that join is associative over binary relations may be written:

```
assert {all p, q, r: univ -> univ | (p.q).r = p.(q.r)}
```

Many such formulas become first order when presented for analysis, since (as here) the quantified variables can be skolemized away. If a formula remains truly higher order, the Alloy Analyzer will warn the user that analysis is likely to be infeasible.

Alloy provides rudimentary support for integers. There is a class of expressions whose values are integers. Integer values may not be

bound to variables in instances, but there is a special class of integer atoms that are associated with primitive integer values, and which may appear in relations that are bound to variables like any other atoms. See Section 8.7 for more details.

7 Types and Overloading

Alloy's type system was designed with different aims from that of a programming language. There is no notion in a modelling language of a 'runtime error', so type soundness is not an issue. Instead, the type system is designed to allow as many reasonable models as possible, without generating false alarms, while still catching prior to analysis those errors that can be explained in terms of the types of declared fields and variables alone.

We expect most users to be able to ignore the subtleties of the type system. Error messages reporting that an expression is ill-typed are never spurious, and always correspond to a real error. Messages reporting failure to resolve an overloaded field reference can always be handled by a small and systematic modification, explained below.

7.1 Type Errors

Three kinds of type error are reported:

- An *arity error* indicates an attempt to apply an operator to an expression of the wrong arity, or to combine expressions of incompatible arity. Examples include: taking the closure of a non-binary relation; restricting a relation to a non-set; taking the union, intersection or difference, or comparing with equality or subset, two relations of different arity.
- A *disjointness error* indicates an expression in which two relations are combined in such a way that the result will always be the empty relation, irrespective of their value. Examples include: taking the intersection of two relations that do not intersect; joining two relations that have no matching elements; and restricting a relation with a set disjoint from it. Applying the overriding operator to disjoint relations also generates a disjointess error, even though the result may not be the empty

relation, since the relations are expected to overlap (a union sufficing otherwise).

- An *irrelevance error* indicates that an expression (usually appearing in a union expression) is redundant, and could be dropped without affecting the value of the enclosing formula. Examples include: expressions such as `(a+b)&c` and formulas such as `c in a+b`, where one of `a` or `b` is disjoint from `c`.

Note that unions of disjoint types *are* permitted, because they might not be erroneous. Thus the expression `(a+b).c`, for example, will be type correct even if `a` and `b` have disjoint types, so long as the type of the leftmost column of `c` overlaps with the types of the righthand columns of both `a` and `b`.

7.2 Field Overloading

Fields of signatures may be overloaded. That is, two distinct signatures may have fields of the same name, so long as the signatures do not represent sets that overlap. Field references are resolved automatically.

Resolution of overloading exploits the full context of an expression, and uses the same information used by the type checker. Each possible resolving of an overloaded reference is considered. If there is exactly one that would not generate a type error, it is chosen. If there is more than one, an error message is generated reporting an ambiguous reference.

Resolution takes advantage of all that is know about the types of the possible resolvents, including arity, and the types of all columns (not only the first). Thus, in contrast to the kind of resolution used for field dereferencing in object-oriented languages (such as Java), the reference to `f` in an expression such as `x.f` can be resolved not only by using the type of `x`, but by using in addition the context in which the entire expression appears. For example, if the enclosing expression were `a+x.f`, the reference `f` could be resolved by the arity of `a`.

If a field reference cannot be resolved, it is easy to modify the expression so that it can be. If a field reference `f` is intended to refer to the field `f` declared in signature `S`, one can replace a reference to `f` by the expression `S <: f`. This new expression has the same mean-

ing, but is guaranteed to resolve the reference, since only the f declared in S will produce a non-empty result. Note that this is *not* a special casting syntax. It relies only the standard semantics of the domain restriction operator.

7.3 Subtypes

The type system includes a notion of subtypes. This allows more errors to be caught, and permits a finer-grained namespace for fields.

The type of any expression is a *union type* consisting of the union of some relation types. A *relation type* is a product of basic types. A basic type is either a signature type, the predefined universal type univ, or the predefined empty type none. The basic types form a lattice, with univ as its maximal, and none as its minimal, element. The lattice is obtained from the forest of trees of declared signature types, augmented with the subtype relationship between top-level types and univ, and between none and all signature types.

The union consisting of no relation types is used in type checking to represent ill-typed expressions, and is distinct from the union consisting of a relation type that is a product of none's (which is used for expressions constructed with the constant none, representing an intentionally empty relation).

The semantics of subtyping is very simple. If one signature is a subtype of another, it represents a subset. The immediate subtypes of a signature are disjoint. Two subtypes therefore overlap only if one is, directly or indirectly, a subtype of the other. The type system computes a type for an expression that is an approximation to its value. Consider, for example, the join

```
e1 . e2
```

where the subexpressions have types

```
e1 : A -> B
e2 : C -> D
```

If the basic types B and C do not overlap, the join gives rise to a disjointness error. Otherwise, one of B or C must be a subtype of the other. The type of the expression as a whole will be A -> D.

No casts are needed, either upwards or downwards. If a field f is declared in a signature S, and sup and sub are respectively variables whose types are a supertype and subtype of S, both sup.f and sub.f will be well typed. In neither case is the expression necessarily empty. In both cases it may be empty: if sup is not in S or f is declared to be partial and sub is outside its domain. On the other hand, if d is a variable whose type D is disjoint from the type of S – for example because both S and D are immediate subtypes of some other signature – the expression d.f will be ill-typed, since it must always evaluate to the empty relation.

7.4 Functions and Predicates

Invocations of functions and predicates are type checked by ensuring that the actual argument expressions are not disjoint from the formal arguments. The types of formals are *not* used to resolve overloading of field names in actual expressions.

The constraints implicit in the declarations of arguments of functions and predicates are conjoined to the body formula when a function or predicate is run. When a function or predicate is invoked, however, these implicit constraints are ignored. You should therefore not rely on such declaration constraints to have a semantic effect; they are intended as redundant documentation. A future version of Alloy may include a checking scheme that determines whether actual expressions have values compatible with the declaration constraints of formals.

7.5 Integers and Type Checking

Only integer expressions take on primitive integer values. The parser distinguishes between relational expressions and integer expressions, so type information is not needed to resolve the overloading of the plus and minus operators (which act as addition and subtraction for integer expressions, and union and difference for relational expressions). In a formula such as

```
#S+S =1
```

the plus symbol will be parsed as a relational operator (and the # operator will be applied to the enture left-hand side), since otherwise the formula as a whole would not be syntactically valid.

The Int type, which represents the predefined signature for integer-carrying objects, is treated by the type system like any other basic type. It is disjoint from all other basic types except for the universal type univ.

7.6 Multiplicity Keywords

Alloy uses the following *multiplicity keywords* shown with their interpretations:

- lone: zero or one;
- one: exactly one;
- some: one or more.

To remember that lone means zero or one, it may help to think of the word as short for 'less than or equal to one'.

These keywords are used in several contexts:

- As quantifiers in quantified formulas: the formula one x: S | F, for example, says that there is exactly one x that satisfies the formula F;
- As quantifiers in quantified expressions: the formula lone e, for example, says that the expression e denotes a relation with containing at most one tuple;
- In set declarations: the declaration x: some S, for example, where S has unary type, declares x to be a set of elements drawn from S that is non-empty;
- In relation declarations: the declaration r: A one -> one B, for example, declares r to be a one-to-one relation from A to B.
- In signature declarations: the declaration one sig S {...}, for example, declares S to be a signature whose set contains exactly one element.

When declaring a set variable, the default is one, so in a declaration x: X in which X has unary type, x will be constrained to be a scalar. In this case, the set keyword overrides the default.

8 Language Features

8.1 Module Structure

The productions discussed in this section are:

module ::= header import* paragraph*
header ::= **module** *[*path*]* moduleId *[***[** sigId,$^+$ **]***]*
import ::= **open** *[*path*]* moduleId *[***[** sigRef,* **]***] [***as** aliasId*]*
paragraph ::= sigDecl | factDecl | funDecl | predDecl | assertDecl
| runCmd | checkCmd
path ::= id / *[*path*]*
sigRef ::= *[*moduleRef*]* sigId | **Int** | **univ**
moduleRef ::= *[*path*]* moduleId *[***[** sigRef,* **]***]* | aliasId
funRef ::= *[*moduleRef*]* funId
predRef ::= *[*moduleRef*]* predId
assertRef ::= *[*moduleRef*]* assertId

An Alloy model consists of one or more *files*, each containing a single *module*. One 'main' module is presented for analysis; it *imports* other modules directly (through its own imports) or indirectly (through imports of imported modules).

A module consists of a header identifying the module, some imports, and some paragraphs:

module ::= header import* paragraph*

A model can be contained entirely within one module, in which case no imports are necessary. A module without paragraphs is syntactically valid but useless.

The paragraphs of a module are signatures, facts, functions, predicates, assertions, run commands and check commands:

paragraph ::= sigDecl | factDecl | funDecl | predDecl | assertDecl
| runCmd | checkCmd

Signatures represent sets and are assigned values in analysis; they therefore play a role similar to static variables in a programming language. Facts, functions and predicates are packagings of constraints. Commands are used to instruct the Alloy Analyzer to perform various analyses. A module exports as components all

paragraphs except for commands; only the commands of the main module are relevant in an analysis.

A module is named by a *path* and a *module identifier*, and may be *parameterized* by one or more signature parameters:

header ::= **module** *[*path*]* moduleId *[***[** sigId**,**$^+$ **]***]*
path ::= id / *[*path*]*

The path must correspond to the directory location of the module's file with respect to a default root directory. A set of root directories may be specified in the Alloy Analyzer, so that libraries and domain-specific models, for example, may be kept in different locations. A module with the module identifier `m` must be stored in the file named `m.als`.

A separate import is needed for each imported module. It gives the path and name of the imported module, instantiations of its parameters (if any), and optionally an alias:

import ::= **open** *[*path*]* moduleId *[***[** sigRef**,*** **]***] [***as** aliasId*]*
sigRef ::= *[*moduleRef*]* sigId | **Int** | **univ**

There must be an instantiating signature parameter for each parameter of the imported module. An instantiating signature may be a type, subtype or subset, or one of the predefined types `Int` and `univ`. If the imported module declares a signature that is an extension of a signature parameter, instantiating that parameter with a subset or with `Int` will be an error.

A single module may be imported more than once. The result is *not* to create multiple copies of the same module, but rather to make a single module available under different names.

A component of an imported module is referred to by its *qualified name*, consisting of the module reference and the component name:

sigRef ::= *[*moduleRef*]* sigId | **Int** | **univ**
funRef ::= *[*moduleRef*]* funId
predRef ::= *[*moduleRef*]* predId
assertRef ::= *[*moduleRef*]* assertId

When a component reference would be ambiguous, it *must* be qualified. Components declared in the same module in which they

are referenced need not be qualified. A module may also be given an *alias* when imported to allow more succinct qualified names. If an alias is declared, the regular module name may not be used.

The module reference may be either the path and module identifier of the imported module along with any instantiating parameters (exactly as it appears in the import statement), or an alias if one was declared in the import:

moduleRef ::= *[*path*]* moduleId *[***[** sigRef,* **]***]* **/** | aliasId **/**

Paragraphs may appear in a module in any order. There is no requirement of definition before use. The order of import statements is also immaterial, even if one provides instantiating parameters to another.

The signature Int is a special predefined signature representing integers, and can be used without an explicit import.

A module may not contain references to components of another module that it does not import, even if that module is imported along with it in another module.

Module names occupy their own name space, and may thus coincide with the names of signatures, paragraphs, arguments or variables without conflict.

8.2 Signature Declarations

The productions discussed in this section are:

sigDecl ::=
  *[***abstract***]* *[*mult*]* **sig** sigID,$^+$ *[***extends** sigRef*]* sigBody
  | *[*mult*]* **sig** sigID,$^+$ **in** sigRef sigBody
sigRef ::= *[*moduleRef*]* sigId | **Int**
sigBody ::= **{** decl,* **}** *[*formulaSeq*]*
formulaSeq ::= **{** formula* **}**
moduleRef ::= *[*path*]* moduleId *[***[** sigRef,* **]***]* | aliasId
mult ::= **lone** | **one** | **some**

A *signature* represents a set of atoms. There are two kinds of signature. A signature declared using the in keyword is a *subset signature*:

sigDecl ::= *[*mult*]* **sig** sigID,$^+$ **in** sigRef sigBody

All other signatures are *type signatures*:

sigDecl ::= *[***abstract***]* *[*mult*]* **sig** sigID,$^+$ *[***extends** sigRef*]* sigBody

A type signature plays the role of a type or subtype in the type system. A type signature that does not extend another signature is a *top-level* signature, and its type is a *top-level type*. A signature that extends another signature is said to be a *subsignature* of the signature it extends, and its type is taken to be a subtype of the type of the signature extended. A signature may not extend itself, directly or indirectly. The type signatures therefore form a type hierarchy whose structure is a forest: a collection of trees rooted in the top-level types.

Top-level signatures represent mutually disjoint sets, and subsignatures of a signature are mutually disjoint. Any two distinct type signatures are thus *disjoint* unless one extends the other, directly or indirectly, in which case they *overlap*.

A subset signature represents a set of elements that is a subset of the union of its *parents*: the signatures listed in its declaration. These may be subset or type signatures. A subset signature may not be extended, and subsets of a signature are not necessarily mutually disjoint. A subset signature may not be its own parent, directly or indirectly. The subset signatures and their parents therefore form a directed acyclic graph, rooted in type signatures. The type of a subset signature is in general a union of top-level types or subtypes, consisting of the parents of the subset that are types, and the types of the parents that are subsets.

An *abstract signature*, marked abstract, is constrained to hold only those elements that belong to one of the signatures that extends it. If there are no extensions, the marking has no effect. The intent is that an abstract signature represents a classification of elements that is refined further by more 'concrete' signatures. If it has no extensions, the abstract keyword is likely an indication that the model is incomplete.

Any multiplicity keyword (with the exception of set, since it has no effect), may be associated with a signature, and constrains the signature's set to have the number of elements specified by the multiplicity.

The body of a signature declaration consists of declarations of *fields*, and an optional *signature fact* constraining the elements of the signature:

sigBody ::= **{** decl,* **}** *[*formulaSeq*]*

A subtype signature *inherits* the fields of the signature it extends, along with any fields that signature inherits. A subset signature inherits the fields of its parent signatures, along with their inherited fields.

A signature may not declare a field whose name conflicts with the name of an inherited field. Moreover, two subset signatures may not declare a field of the same name if their types overlap. This ensures that two fields of the same name can only be declared in disjoint signatures, and there is always a context in which two fields of the same name can be distinguished. If this were not the case, some overloadings would never be resolvable.

Like any other fact, the signature fact is a constraint that always holds. Unlike other facts, however, a signature fact is implicitly quantified over the signature set. Given the signature declaration

sig S {...} { *F* }

the signature fact *F* is interpreted as if one had written an explicit fact

fact { **all** this: S | *F′* }

where *F′* is like *F*, but has each reference to a field f of S (whether declared or inherited) replaced by this.f. Prefixing a field name with the special symbol @ preempts this implicit expansion.

Declaring multiple signatures at once in a single signature declaration is equivalent to declaring each individually. Thus the declaration

sig A, B **extends** C {f: D}

for example, introduces two subsignatures, A and B, of C, and for each declares a field f.

8.3 Declarations

The productions discussed in this section are:

```
decl ::= [part | disj] varId,+ : declExpr
declExpr ::=
   [mult] expr
   | expr [mult] -> [mult] expr
mult ::= lone | one | some
```

The same declaration syntax is used for fields of signatures, arguments to functions and predicates, and quantified variables – all of which we shall here refer to as *variables*. The interpretation for fields, which is slightly different, is explained second.

A declaration introduces one or more variables, and constrains their values and type:

```
decl ::= [part | disj] varId,+ : declExpr
```

A declaration has two effects:

- Semantically, it constrains the *value* a variable can take. The relation denoted by the variable (on the left) is constrained to be a subset of the relation denoted by the declaration expression (on the right). When more than one variable is declared at once, the keywords disj and part may be used. The keyword disj constrains the declared variables to be mutually disjoint. The keyword part constrains them additionally to form a partition of the relation denoted by the declaration expression. Multiplicity constraints, explained below, constrain the value of a variable further.
- For the purpose of type checking, a declaration gives the variable a *type*. A type is determined for the declaration expression, and that type is assigned to the variable. Any variable that appears in the declaration expression must have been declared already, either earlier in the sequence of declarations in which this declaration appears, or earlier elsewhere. For a quantified variable, this means within an enclosing quantifier; for a field of a signature, this means that the field is inherited; for a function or predicate argument, this means in the argument declarations of the enclosing function or predicate.

Note that the declaration expression of a field declaration in a signature may not refer to fields declared in other signatures, unless they are inherited.

The declaration expression is an arbitrary expression. If the expression denotes a set (that is, a unary relation), it may be prefixed by a multiplicity keyword:

declExpr ::= *[*mult | **set***]* expr
mult ::= **lone** | **one** | **some**

If the keyword is omitted, the declared variable is constrained by default to be a *scalar* (that is, to be a singleton set). The keyword `set` eliminates this constraint; `lone` weakens it to allow the variable to denote a 'loner' or 'option': either a singleton set or the empty set; `some` constrains the variable to denote a non-empty set; and `one` has no effect, being equivalent to omission.

If the expression denotes a relation of higher arity, the multiplicity keywords may not be used as a prefix. If the expression is formed with the arrow operator, the arrow itself may be elaborated with multiplicity keywords:

declExpr ::= expr *[*mult*]* **->** *[*mult*]* expr
mult ::= **lone** | **one** | **some**

If the declaration expression has the form `e1` *m*`->`*n* `e2`, where *m* and *n* are multiplicity keywords, the declaration imposes a *multiplicity constraint* on the declared variable. An arrow expression of this form denotes the relation whose tuples are concatenations of the tuples in `e1` and the tuples in `e2`. If the marking *n* is present, the relation denoted by the declared variable is required to contain, for each tuple `t1` in `e1`, *n* tuples that begin with `t1`. If the marking *m* is present, the relation denoted by the declared variable is required to contain, for each tuple `t2` in `e2`, *m* tuples that end with `t2`. The markings are interpreted as follows:

- `lone` means zero or one;
- `one` means exactly one;
- `some` means one or more.

When the expressions `e1` and `e2` are unary, these reduce to familiar notions. For example, a declaration expression of the form `X ->one Y` makes the variable a total function from `X` to `Y`; the expression `X ->lone Y` makes it an partial function; and `X one->one Y` makes it a bijection.

Declarations within a signature have essentially the same interpretation. But for a field `f`, the declaration constraints apply not to

f itself but to this.f: that is, to the value obtained by dereferencing an element of the signature with f. Thus, for example, the declaration

sig S {f: e}

does not constrain f to be a subset of e (as it would if f were a regular variable), but rather implies

all this: S | this.f **in** e

Moreover, any field appearing in e is expanded according to the rules of signature facts (Section 8.2). A similar treatment applies to multiplicity constraints and disj/part. In this case, for example, if e denotes a unary relation, the implicit multiplicity constraint will make this.f a scalar, so that f itself will denote a total function on S.

Type checking of fields has the same flavour. The field f is not assigned the type e, but rather the type of the expression S -> e. That is, the domain of the relation f has the type S, and this.f has the same type as e.

8.4 Constraint Paragraphs

The productions discussed in this section are:

factDecl ::= **fact** *[*factId*]* formulaSeq
predDecl ::= **pred** *[*sigRef **::***]* predId **(** decl,* **)** formulaSeq
funDecl ::= **fun** *[*sigRef **::***]* funId **(** decl,*) **:** declExpr **{** expr **}**
assertDecl ::= **assert** *[*assertId*]* formulaSeq
formulaSeq ::= **{** formula* **}**
formula ::= … | *[*expr **::***]* predRef **(** expr,* **)**
expr ::= … | *[*expr **::***]* funRef **(** expr,* **)**

A *fact* is a constraint that always holds. A *predicate* is a template for a constraint that can be instantiated in different contexts. A *function* is a template for an expression. An *assertion* is a constraint that is intended to follow from the facts of a model; it is thus an intentional redundancy. Assertions can be checked by the Alloy Analyzer; functions and predicates can be simulated.

A fact can be named for documentation purposes. An assertion can be named or anonymous, but since a command to check an as-

sertion must name it, an anonymous assertion cannot be checked. Functions and predicates must always be named.

A fact consists of an optional name and a constraint, given as a sequence of formulas, which are implicitly conjoined:

factDecl ::= **fact** *[*factId*]* formulaSeq

A predicate declaration consists of the name of the predicate, some argument declarations, and a constraint, given as a sequence of formulas, which are implicitly conjoined:

predDecl ::= **pred** *[*sigRef **::***]* predId **(** decl,* **)** formulaSeq

The argument declarations may include a first argument declared anonymously. When a predicate is declared in the form

pred S::f (…) {…}

the first argument is taken to be a scalar of signature S, which is referred to within the body of the predicate using the keyword this, as if the declaration had been written

pred f (this: S, …) {…}

A function declaration consists of the name of the predicate, some argument declarations, and an expression:

funDecl ::= **fun** *[*sigRef **::***]* funId **(** decl,*) **:** declExpr **{** expr **}**

The argument declarations include a declaration expression for the result of the function, corresponding to the value of the expression. The first argument may be declared anonymously, exactly as for predicates.

A predicate may be invoked as a formula by providing an expression for each argument:

formula ::= *[*expr **::***]* predRef **(** expr,* **)**

A function likewise may be invoked as an expression by providing an expression for each argument:

expr ::= *[*expr **::***]* funRef **(** expr,* **)**

Invocation can be viewed as textual inlining. An invocation of a predicate gives a formula which is obtained by taking the formula of the predicate's body, and replacing the formal arguments by the corresponding expressions of the invocation. Likewise, invocation

of a function gives an expression obtained by taking the expression of the function's body, and replacing the formal arguments of the function by the corresponding expressions of the invocation. Recursive invocations are not currently supported.

A function or predicate invocation may present its first argument in receiver position. So instead of writing

```
p (a, b, c)
```

for example, one can write

```
a::p (b, c)
```

The form of invocation is not constrained by the form of declaration. Although often a function or predicate will be both declared with an anonymous receiver argument and used with receiver syntax, this is not necessary. The first argument may be presented as a receiver irrespective of the format of declaration, and the first argument may be declared anonymously irrespective of the format of use. In particular, it can be convenient to invoke a function or predicate in receiver form when the first argument is not a scalar, even though it cannot be declared with receiver syntax in that case.

Within a module, no two constraint paragraphs may be declared with the same name, nor may a constraint paragraph have the same name as a signature.

8.5 Commands

The productions discussed in this section are:

```
runCmd ::=
   [commandId :] run funRef [scope]
   [commandId :] run predRef [scope]
checkCmd ::= [commandId :] check assertRef [scope]
scope ::= for number
   | for [number but] typescope,+
typescope ::= [exactly] number scopeable
scopeable ::= sigRef | int
sigRef ::= [moduleRef] sigId | Int | univ
```

A command is an instruction to the Alloy Analyzer to perform an analysis. Analysis involves constraint solving. A run command

causes the analyzer to search for an *instance* that witnesses the consistency of a function or a predicate. A check command causes it to search for a *counterexample* showing that an assertion does not hold.

A command to run a function or predicate consists of an optional command name, the keyword run, a reference to the function or predicate, and, optionally, a scope specification:

runCmd ::=
  *[*commandId **:***]* **run** funRef *[*scope*]*
  *[*commandId **:***]* **run** predRef *[*scope*]*

Similarly, a command to check an assertion consists of an optional command name, the keyword check, a reference to the assertion, and, optionally, a scope specification:

checkCmd ::= *[*commandId **:***]* **check** assertRef *[*scope*]*

The command name is used in the user interface of the Alloy Analyzer (or at the command line) to select the command to be executed. In the graphical user interface, the command is selected from a pop-up menu; the only reason for the command name is to allow commands to be more easily recognized when there are many commands for the same assertion, function or predicate. No two commands in a module may have the same command names.

As explained in Section 6, analysis always involves solving a constraint. For a predicate with body formula P, the constraint solved is

P **and** F **and** D

where F is the conjunction of all facts, and D is the conjunction of all declaration constraints, including the declarations of the predicate's arguments. Note that when the predicate's body is empty, the constraint is simply the facts and declaration constraints of the model. An empty predicate is often a useful starting point in analysis to determine whether the model is consistent, and, if so, to examine some of its instances.

For a function named f whose body expression is E, the constraint solved is

f = E **and** F **and** D

where F is the conjunction of all facts, and D is the conjunction of all declaration constraints, including the declarations of the function arguments. The variable f stands for the value of the expression.

Note that the declaration constraints of a predicate or function are used when that function or predicate is run, but are ignored when the predicate or function is invoked.

For an assertion whose body formula is A, the constraint solved is

F **and** D **and not** A

namely the negation of

F **and** D **implies** A

where F is the conjunction of all facts, and D is the conjunction of all declaration constraints.

An instance or counterexample found by the analyzer will assign values to the following variables:

- The signatures and fields of the model;
- For an instance of a predicate or function, the arguments of the function or predicate, one of which will be named this if the first argument is declared anonymously;
- For an instance of function, a variable denoting the value of the expression, with the same name as the function itself.

The analyzer may also give values to skolem constants as witnesses for existential quantifications. Whether it does so, and whether existentials inside universals are skolemized, depends on preferences set by the user.

The search for an instance is conducted within a *scope*, which is specified as follows:

```
scope ::= for number
   | for [number but] typescope,+
typescope ::= [exactly] number scopeable
scopeable ::= sigRef | int
sigRef ::= [moduleRef] sigId | Int | univ
```

The *scope specification* of a command places bounds on the sizes of the sets assigned to type signatures, thus making the search finite. Only type signatures are involved; subset signatures may not

be bounded in a scope specification. For the rest of this section, 'signature' should be read as synonymous with 'type signature.'

The bounds are determined as follows:

- If no scope specification is given, a default scope of 3 elements is used: each top-level signature is constrained to represent a set of at most 3 elements.
- If the scope specification takes the form `for` *N*, a default of *N* is used instead.
- If the scope specification takes the form `for` *N* `but` …, every signature listed following `but` is constrained by its given bound, and any top-level signature whose bound is not given implicitly is bounded by the default *N*.
- Otherwise, for an explicit list without a default, each signature listed is constrained by the given bound.

Implicit bounds are determined as follows:

- If an abstract signature has no explicit bound, but its subsignatures have bounds, implicit or explicit, its bound is the sum of those of its subsignatures.
- If an abstract signature has a bound, explicit or by default, and all but one of its subsignatures have bounds, implicit or explicit, the bound of the remaining subsignature is the difference between the abstract signature's bound and the sum of the bounds of the other subsignatures.
- A signature declared with the multiplicity keyword `one` has a bound of 1.
- If an implicit bound cannot be determined for a signature by these rules, the signature has no implicit bound.

If a scope specification uses the keyword `exactly`, the bound is taken to be both an upper and lower bound on the cardinality of the signature. The rules for implicit bounds are adjusted accordingly. For example, an abstract signature whose subsignatures are constrained exactly will likewise be constrained exactly.

The scope specification must be:

- *consistent*: at most one bound must be associated with any signature, implicitly, explicitly or by default;
- *complete*: every top-level signature must have a bound;
- *uniform*: a signature without a bound may not have a subsignature that has a bound.

By default, the predefined signature Int is limited to 3 elements, so that there may be at most 3 integer objects appearing in an instance or counterexample. The bound on the integer values represented by these integer objects, and on the values of integer expressions, may be altered by assigning a bound to int. A bound of k for int limits integer values to be between 0 and 2^k-1. Its default is 5, so integers by default range from 0 to 31.

8.6 Expressions

The productions discussed in this section are:

```
expr ::= [@] varId | sigRef | this |
   | none | univ | iden
   | unOp expr | expr binOp expr | expr[ expr ]
   | { decl,+ | [formula] }
   | let letDecl,+ | expr
   | if formula then expr else expr
   | ( expr )
letDecl ::= varId = expr
binOp ::= + | - | & | . | -> | <: | :> | ++
unOp ::= ~ | * | ^
```

There are two kinds of expression in Alloy: relational expressions, and integer expressions. When mentioned without qualification, the term 'expression' refers to a relational expression.

Every relational expression denotes a relation. A set is represented as a relation of arity one, and a scalar is represented as a singleton set. A tuple is a singleton relation.

Alloy's analysis involves finding solutions to constraints. For any candidate instance that may be a solution to the constraint, each expression of the constraint has a value given by the instance's bindings of values to variables.

An expression may consist simply of a variable name, signature reference, or the special argument this:

```
expr ::= [@] varId | sigRef | this |
```

If the variable denotes a field name, its value is the value bound to that field in the instance being evaluated. In contexts in which field names are implicitly dereferenced – that is, in signature dec-

laration expressions and signature facts – the prefix @ preempts dereferencing (see Section 8.2). If there is more than one field of the given name, the reference is resolved, or rejected if ambiguous (see Section 7).

If a variable denotes a quantified or let-bound variable, its value is determined by the binding. If the variable is an argument of a function or predicate, the analysis at hand must be a run of that function or predicate (since if the function or predicate is invoked, its meaning is obtained by inlining and the argument has been replaced) and the variable's value is bound speculatively to each possible value during search.

An expression may be a relational constant:

expr ::= **none** | **univ** | **iden**

The three constants none, univ and iden denote respectively: the empty unary relation (that is, the set containing no elements), the universal unary relation (the set containing every element) and the identity relation (the binary relation that relates every element to itself).

Note that univ and iden are interpreted over the universe of all atoms. So a formula such as

iden in r

will be unsatisfiable unless the relation r has type univ -> univ. To say that r is a reflexive relation, you might write instead

t <: **iden in** r

for example, where r has type t -> t.

An expression may be a compound expression:

expr ::= un0p expr | expr bin0p expr | expr**[** expr **]**
bin0p ::= **+** | **-** | **&** | **.** | **->**
un0p ::= **~** | ***** | **^**

The value of a compound expression is obtained from the values of its constituents by applying the operator given. The meanings of the operators are as follows:

- ~e: transpose of e;
- ^e: transitive closure of e;

- *e: reflexive-transitive closure of e;
- e1 + e2: union of e1 and e2;
- e1 - e2: difference of e1 and e2;
- e1 & e2: intersection of e1 and e2;
- e1 . e2: join of e1 and e2;
- e2 [e1]: join of e1 and e2;
- e1 -> e2: product of e1 and e2;
- e2 <: e1: domain restriction of e1 to e2;
- e1 :> e2: range restriction of e1 to e2;
- e1 ++ e2: relational override of e1 by e2.

For the first three (the unary operators), e is required to be binary. For the set theoretic operations (union, difference and intersection) and for relational override, the arguments are required to have the same arity. For the restriction operators, the argument e2 is required to be a set.

Note that e1[e2] is equivalent to e2.e1, but the dot and square brackets operators have different precedence.

The *transpose* of a relation is its mirror image: the relation obtained by reversing each tuple. The *transitive closure* of a relation is the smallest enclosing relation that is transitive (that is, relates a to c whenever there is a b such that it relates a to b and b to c). The *reflexive-transitive closure* of a relation is the smallest enclosing relation that is transitive and reflexive (that is, includes the identity relation).

The union, difference and intersection operators are the standard set theoretic operators, applied to relations viewed as sets of tuples. The *union* of e1 and e2 contains every tuple in e1 or in e2; the *intersection* of e1 and e2 contains every tuple in both e1 and in e2; the *difference* of e1 and e2 contains every tuple in e1 but not in e2.

The *join* of two relations is the relation obtained by taking each combination of a tuple from the first relation and a tuple from the second relation, and if the last element of the first tuple matches the first element of the second tuple, including the concatenation of the two tuples, omitting the matching elements.

The *product* of two relations is the relation obtained by taking each combination of a tuple from the first relation and a tuple from the second relation, and including their concatenation.

The *domain restriction* of e1 to e2 contains all tuples in e1 that start with an element in the set e2. The *range restriction* of e1 to e2 contains all tuples in e1 that end with an element in the set e2. These operators are especially handy in resolving overloading (see Section 7).

The *relational override* of e1 by e2 contains all tuples in e2, and additionally, any tuples of e1 whose first element is not the first element of a tuple in e2.

An expression may be a *comprehension expression*:

expr ::= **{** decl,$^+$ **|** *[*formula*]* **}**

The expression

{x1: e1, x2: e2, ... | F}

denotes the relation obtained by taking all tuples x1 -> x2 -> ... in which x1 is drawn from the set e1, x2 is drawn from the set e2, etc., and the formula F holds. The expressions e1, e2, etc., must be unary, and may not be prefixed by multiplicity keywords. More general declaration forms are not permitted, except for the use of the disj and part keywords.

An expression may be a *let expression*:

expr ::= **let** letDecl,$^+$ **|** expr
letDecl ::= varId **=** expr

The expression

let v1 = e1, v2 = e2, ... | e

is equivalent to the expression e, but with each bound variable v1, v2, etc. replaced by its assigned expression e1, e2, etc. Variables appearing in let declaration expressions must have been previously declared. Recursive bindings are not permitted.

An expression may be an *if expression*:

expr ::= **if** formula **then** expr **else** expr

The expression

if F **then** e1 **else** e2

has the value of expression e1 when the formula F is true, and the value of expression e2 otherwise.

The meaning of an *invocation expression*

expr ::= *[*expr **::***]* funRef **(** expr,* **)**

is explained in Section 8.4.

The meaning of the *Integer expression*

expr ::= **Int** intExpr

is explained in Section 8.7.

An expression may be parenthesized to force a particular ordering of application of operators:

expr ::= **(** expr **)**

8.7 Integers

The productions discussed in this section are:

formula ::= intExpr *[*neg*]* intCompOp intExpr
expr ::= **Int** intExpr
intExpr ::= number | **#** expr | **sum** expr | **int** expr
| **if** formula **then** intExpr **else** intExpr
| intExpr intOp intExpr
| **let** letDecl,… **|** intExpr
| **sum** decl,+ **|** intExpr
| **(** intExpr **)**
intOp ::= **+** | **-**
intCompOp ::= **<** | **>** | **=** | **=<** | **>=**

There are two kinds of integers in Alloy. The predefined signature Int denotes a set of *integer-carrying objects* that may appear as atoms in relations. Integer operators may not be applied to these objects directly. Integer expressions are distinguished syntactically from relational expressions, and have *primitive integer values* which may be combined and compared using arithmetic operators. Primitive integer values may not appear as atoms in relations, and cannot be quantified over.

Distinct integer objects never carry the same primitive integer value. So the following assertion always holds:

assert IntegersCanonical {**no disj** i, j: **Int** | **int** i = **int** j}

A primitive integer value may be obtained from a relational expression whose value is a set of integer objects:

intExpr ::= **sum** expr | **int** expr

Both integer expressions **int** e and **sum** e have an integer value that is the sum of the integer values associated with integer objects in the set denoted by the relational expression e. There is no semantic difference between the two. The intent is that sum be used to indicate explicitly that the expression is expected not to be a singleton. Usually, the int operator will be applied to an expression denoting a single Integer object, but it is defined over a set of Integer objects so that it always has a value.

A primitive integer value may be obtained from a relational expression of any type using a *cardinality expression*:

intExpr ::= **#** expr

The integer expression #e has an integer value corresponding to the cardinality of e – that is, the number of tuples in the relation denoted by the relational expression e.

A *numeric constant* may be used as an integer expression:

intExpr ::= number

A numeric constant is a sequence of one or more digits, of which the first is not zero.

Integers may be combined using standard arithmetic operators for addition and subtraction:

intExpr ::= intExpr intOp intExpr
intOp ::= **+** | **-**

The integer expression i + j evaluates to the sum of the values of the integer expressions i and j; the integer expression i - j evaluates to the value of the integer expression i minus the value of the integer expression j. Note that the plus and minus symbols are overloaded: they are treated as arithmetic operators within integer expressions, and as relational operators within relational expressions.

A *sum expression* computes the sum of the values of an integer expression over a range of bindings:

intExpr ::= **sum** decl,$^+$ **|** intExpr

The integer expression

sum x: X, y: Y, … | ie

evaluates to the sum of the values that the integer expression ie can take for all distinct bindings of the variables x, y, etc. The most general declaration forms are permitted, although analysis may not be feasible when the bindings are not first order.

If-then-else and let can be applied to integer expressions:

intExpr ::=
if formula **then** intExpr **else** intExpr
| **let** letDecl,… **|** intExpr

with the same meaning as for relational expressions, but with integer values instead.

Integer valued expressions can be compared:

formula ::= intExpr *[*neg*]* intCompOp intExpr
intCompOp ::= **<** | **>** | **=** | **=<** | **>=**

The meaning of the comparison operators is as follows:

- The formula i = j is true when the integer expressions i and j have the same value;
- The formula i < j is true when i is less than j;
- The formula i > j is true when i is greater than j;
- The formula i =< j is true when i is less than or equal to j;
- The formula i >= j is true when i is greater than or equal to j.

The 'less than or equal to' operator is written unconventionally with the equals symbol first so that it does not have the appearance of an arrow, which might be confused with a logical implication.

A formula in which the comparison operator is negated

e1 **not** *op* e2

is equivalent to the formula obtained by moving the negation outside:

not e1 *op* e2

The negation operators ! and not have the same meaning.

Integer objects are obtained from integer values with the Int operator:

expr ::= **Int** intExpr

The expression **Int** ie denotes the Integer object associated with the value of the integer expression ie; it is equivalent to

{i: **Int** | **int** i = ie}

It is possible that, in a particular analysis, the scope is too small to provide such an integer. In that case, Int ie denotes the empty set. Note that because no two integer-carrying objects hold the same integer value, it will never denote a set of more than one object.

8.8 Formulas

The productions discussed in this section are:

formula ::=
  quantifier expr
  | expr *[*neg*]* compOp expr
  | neg formula | formula logicOp formula
  | formula thenOp formula *[*elseOp formula*]*
  | quantifier decl,$^+$ formulaBody
  | **let** letDecl,$^+$ formulaBody
  | expr **:** declExpr
  | formulaSeq
  | **(** formula **)**
formulaBody ::= formulaSeq | **|** formula
formulaSeq ::= **{** formula* **}**
letDecl ::= varId **=** expr
thenOp ::= **implies** | **=>**
elseOp ::= **else** | **,**
neg ::= **not** | **!**
logicOp ::= **&&** | **||** | **iff** | **<=>** | **and** | **or**
quantifier ::= **all** | **no** | mult
mult ::= **lone** | **one** | **some**
compOp ::= **in** | **:** | **=**
declExpr ::=
  *[*mult | **set***]* expr
  | expr *[*mult*]* **->** *[*mult*]* expr

Elementary formulas are formed by applying quantifiers to relational expressions, or by comparing relational or integer expressions.

A *quantified expression* takes the form

```
formula ::= quantifier expr
quantifier ::= all | no | mult
mult ::= lone | one | some
```

Its meaning depends on the quantifier chosen:

- The formula **no** e is true when e evaluates to a relation containing no tuple.
- The formula **some** e is true when e evaluates to a relation containing one or more tuple.
- The formula **lone** e is true when e evaluates to a relation containing at most one tuple.
- The formula **one** e is true when e evaluates to a relation containing exactly one tuple.

The formula **all** e is rejected by a static semantic check: it has no meaning.

A *comparison formula* takes the form

```
formula ::= expr [neg] compOp expr
compOp ::= in | =
```

Its meaning depends on the comparison operator:

- The formula `e1` **in** `e2` is true when the relation that `e1` evaluates to is a subset of the relation that `e2` evaluates to.
- The formula `e1 = e2` is true when the relation that `e1` evaluates to to the same relation as `e2`.

Note that relational equality is extensional: two relations are equal when they contain the same tuples.

A formula in which the comparison operator is negated

```
e1 not op e2
```

is equivalent to the formula obtained by moving the negation outside:

```
not e1 op e2
```

The negation operators **!** and **not** have the same meaning.

Comparisons on integer expressions are covered in Section 8.7.

A *negated formula* takes the form

```
formula ::= neg formula
neg ::= not | !
```

The formula **not** F is true when the formula F is false, and vice versa. The negation operators **not** and **!** are interchangeable.

A *compound formula* combines smaller formulas with logical operators:

```
formula ::=
   formula logicOp formula
   | formula thenOp formula [elseOp formula]
logicOp ::= && | || | iff | <=> | and | or
thenOp ::= implies | =>
elseOp ::= else | ,
```

The meaning of the logical operators is as follows:

- The formula F **and** G is true when F is true and G is true.
- The formula F **or** G is true when one or both of F and G are true.
- The formula F **iff** G is true when F and G are both false or both true.
- The formula F **implies** G is true when F is false or G is true.
- The formula F **implies** G **else** H is true when both F and G are true, or when F is false and H is true.

The logical operators may be written interchangeably as symbols: **&&** for **and**, **||** for **or**, **=>** for **implies**, **<=>** for **iff**, and a comma (,) for **else**.

A *formula sequence* is a sequence of formulas enclosed in curly braces:

```
formula ::= formulaSeq
formulaSeq ::= { formula* }
```

The formula

```
{ F G H ... }
```

is equivalent to the conjunction

```
F and G and H and ...
```

If the sequence contains no formulas, the formula is true.

A *quantified formula* consists of one or more declarations and a body:

```
formula ::= quantifier decl,+ formulaBody
formulaBody ::= formulaSeq | | formula
formulaSeq ::= { formula* }
quantifier ::= all | no | mult
mult ::= lone | one | some
```

It makes no difference whether the formula body is a single formula preceded by a vertical bar, or a formula sequence. The two forms are provided so that the vertical bar can be omitted when the body is a sequence of formulas. Some users prefer to use the bar in all cases, writing for example:

```
all x: X | { F }
```

Others prefer never to use the bar, and use the curly braces even when the formula sequence consists of only a single formula:

```
all x: X { F }
```

These forms are all acceptable and are interchangeable.

The meaning of the formula depends on the quantifier:

- The formula **all** x: e | F is true when the formula F is true for all bindings of the variable x.
- The formula **no** x: e | F is true when the formula F is true for no bindings of the variable x.
- The formula **some** x: e | F is true when the formula F is true for one or more bindings of the variable x.
- The formula **sole** x: e | F is true when the formula F is true for at most one binding of the variable x.
- The formula **one** x: e | F is true when the formula F is true for exactly one binding of the variable x.
- The formula **two** x: e | F is true when the formula F is true for exactly two bindings of the variable x.

The range and type of the bound variable is determined by its declaration (see Section 8.3). In a sequence of declarations, each declared variable may be bound by the declarations or previously declared variables. For example, in the formula

```
all x: e, y: S - x | F
```

the variable x varies over the values of the expression e (assumed to represent a set), and the variable y varies over all elements of the set S except for x. When more than one variable is declared, the quantifier is interpreted over bindings of all variables. For example,

one x: X, y: Y | F

is true when there is exactly one binding that assigns values to x and y that makes F true. So although a quantified formula with multiple declarations may be regarded, for some quantifiers, as a shorthand for nested formulas each with one declaration, this is not in general true. Thus

all x: X, y: Y | F

is short for

all x: X | **all** y: Y | F

but

one x: X, y: Y | F

is not short for

one x: X | **one** y: Y | F

A quantified formula may be higher-order: that is, it may bind non-scalar values to variables. Whether the formula is analyzable will depend on whether it can be skolemized by the Analyzer, and if not, how large the scope is.

A *let formula* allows a variable to be introduced, to highlight an import subexpression or make the formula shorter by factoring out a repeated subexpression:

formula ::= **let** letDecl,$^+$ formulaBody
letDecl ::= varId **=** expr

The formula

let x1 = e1, x2 = e2, ... | F

is equivalent to the formula F with each occurrence of the bound variable x1 replaced by the expression e1, x2 by e2, etc. Like all declarations, let declarations are interpreted in order, and may not be recursive.

Predicate invocation is discussed in Section 8.4.

A *declaration formula* allows a multiplicity constraint to be placed on an expression:

```
formula ::= declFormula
declFormula ::= expr : declExpr
declExpr ::=
   [mult | set] expr
   | expr [mult] -> [mult] expr
```

Declaration formulas are useful for two reasons. First, they allow multiplicity constraints to be placed on arbitrary expressions, where declarations themselves only allow them to be placed on variables. Thus,

```
p.q : t one->one t
```

for example, says that the join of `p` and `q` is a bijection. Second, they allow additional multiplicity constraints to be expressed that cannot be expressed in declarations. For example, the relation `r` of type `A -> B` can be declared as a field of `A`:

```
sig A {r: set B}
```

Since the declaration constraints apply to the relations `this.r`, they cannot constrain the multiplicity of the relation from `B`'s perspective. To say that `r` maps at most one `A` to each `B`, one could add as a fact the declaration formula

```
r: A lone-> B
```

Another deficiency of declarations that can be overcome is that they only allow multiplicities around one arrow to be given. For a relation `p` of type `A -> B -> C`, a declaration of the form

```
all r: A ->some (B -> C) | ...
```

makes `r` total on `A`. The constraint that `R` maps a pair from `A -> B` to each element of `C` cannot be expressed in this declaration because it requires a different parsing of the expression, associating the arrows to the left rather than the right. To express this constraint, one could use a declation formula like this:

```
all r: A ->some (B -> C) | r: (A -> B) some-> C  => ...
```

A formula may be parenthesized to force a particular ordering of application of operators:

formula ::= **(** formula **)**