Appendix B: Alloy Language Reference

B.1 Lexical Issues

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

- backslash \
- backquote `

and, of the ASCII nonprinting 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 return and linefeed.

The nonalphanumeric symbols (including hyphen) are used as operators or for punctuation, with the exception of

- dollar sign *$*;
- percent sign *%*;
- question mark *?*;
- exclamation point *!*;
- underscore _;
- single and double quote marks (*'* and *"*).

Dollar, percent and 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 replace simple quote marks with typographic ones automatically.

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

Noncomment text is broken into tokens by the following separators:

- whitespace (space, tab, linebreak);
- nonalphanumeric 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, 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 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
iff	implies	in	Int	let
lone	module	no	none	not
one	open	or	pred	run
set	sig	some	sum	univ

B.2 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, with the innermost binding taking precedence.

B.3 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, potentially ambiguous symbols—namely parentheses, square brackets, star, plus and the vertical bar—are set in bold type when they are to be interpreted as terminals rather than as meta symbols. The string *name* represents an identifier and *number* represents a numeric constant, according to the lexical rules above (section B.1).

alloyModule ::= *[*moduleDecl*]* import* paragraph*

moduleDecl ::= **module** qualName *[***[**name,$^+$**]***]*

import ::= **open** qualName *[***[**qualName,$^+$**]***] [***as** name*]*

paragraph ::= sigDecl | factDecl | predDecl | funDecl
| assertDecl | cmdDecl

sigDecl ::= *[***abstract***]* *[*mult*]* **sig** name,$^+$ *[*sigExt*]* **{** decl,* **}** *[*block*]*

sigExt ::= **extends** qualName | **in** qualName *[***+** qualName*]**

mult ::= **lone** | **some** | **one**

decl ::= *[***disj***]* name,$^+$ **:** *[***disj***]* expr

factDecl ::= **fact** *[*name*]* block

predDecl ::= **pred** *[*qualName **.***]* name *[*paraDecls*]* block

funDecl ::= **fun** *[*qualName **.***]* name *[*paraDecls*]* **:** expr **{** expr **}**

paraDecls ::= **(** decl,* **)** | **[** decl,* **]**

assertDecl ::= **assert** *[*name*]* block

cmdDecl ::= *[*name **:***]* *[***run**|**check***]* *[*qualName|block*]* *[*scope*]*

scope ::= **for** number *[***but** typescope,$^+$*]* | **for** typescope,$^+$

typescope ::= *[***exactly***]* number qualName

expr ::= const | qualName | **@**name | **this**
| unOp expr | expr binOp expr | expr arrowOp expr
| expr **[** expr,* **]**
| expr *[***!**|**not***]* compareOp expr
| expr *(***=>**|**implies***)* expr **else** expr
| **let** letDecl,$^+$ blockOrBar
| quant decl,$^+$ blockOrBar
| **{** decl,$^+$ blockOrBar **}**
| **(** expr **)** | block

const ::= *[***-***]* number | **none** | **univ** | **iden**

unOp ::= **!** | **not** | **no** | mult | **set** | **#** | **~** | ***** | **^**

binOp ::= **||** | **or** | **&&** | **and** | **<=>** | **iff** | **=>** | **implies** |
& | **+** | **-** | **++** | **<:** | **:>** | **.**

arrowOp ::= *[*mult|**set***]* **->** *[*mult|**set***]*

compareOp ::= **in** | **=** | **<** | **>** | **=<** | **>=**

letDecl ::= name **=** expr

block ::= **{** expr* **}**

blockOrBar ::= block | bar expr
bar ::= **|**

quant ::= **all** | **no** | **sum** | mult

qualName ::= *[***this/***]* *(*name **/***)** name

The grammar does not distinguish relation-valued expressions from boolean-valued expressions (that is, 'constraints' or 'formulas'). Nevertheless, the two categories are in fact disjoint. Only the boolean operators (such as *&&*, denoting conjunction), and not the relational operators (such as *&*, denoting intersection) can be applied to boolean expressions.

B.4 Precedence and Associativity

Expression operators bind most tightly, in the following precedence order, tightest first:

- unary operators: ~, ^ and *;
- dot join: **.** ;
- box join: *[]*;
- restriction operators: *<:* and *:>*;
- arrow product: ->;
- intersection: *&*;
- override: ++;
- cardinality: *#*;
- union and difference: + and -;
- expression quantifiers and multiplicities: *no*, *some*, *lone*, *one*, *set*;
- comparison negation operators: *!* and *not*;
- comparison operators: *in*, =, <, >, =, =<, =>.

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

Logical operators are bound at lower precedence, as follows:

- negation operators: *!* and *not*;
- conjunction: *&&* and *and*;
- implication: =>, *implies*, and *else*;
- bi-implication: <=>, *iff*;
- disjunction: || and *or*;
- let and quantification operators: *let*, *no*, *some*, *lone*, *one* and *sum*.

All binary operators associate to the left, with the exception of implication, which associates to the right. 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 constraint

```
p => q => r else s
```

for example, is parsed as

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

B.5 Semantic Basis

B.5.1 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 is 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 subsection B.7.6 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.

B.5.2 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, tuples or scalars. A set is simply a unary relation; a tuple is a singleton relation; and a scalar is a singleton, unary relation. The type system distinguishes sets from relations because they have different arity, but does not distinguish tuples and scalars from non-singleton relations.

There is no function application operator; relational join is used in its place. For example, given a relation *f* that is functional, and *x* and *y* constrained to be scalars, the constraint

```
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 constraint 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 outside the domain of *f*, the expression *x.f* will evaluate to the empty set, and since *y* is a scalar (that is, a singleton set), the constraint 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 constraint

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

has *x* range over scalar values: that is, the constraint *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 constraints become first order when presented for analysis, since (as here) the quantified variables can be Skolemized away. If a constraint remains truly higher order, the Alloy Analyzer will warn the user that analysis is likely to be infeasible.

B.6 Types and Overloading

Alloy's type system was designed with a different aim from that of a programming language. There is no notion in a modeling 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 a failure to resolve an overloaded field reference can always be handled by a small and systematic modification, explained below.

B.6.1 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 nonbinary relation; restricting a relation to a non-set (that is, a relation that does not have an arity of one); taking the union, intersection, or difference, or comparing with equality or subset, two relations of unequal 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 its domain or range. 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).
- A *redundancy error* indicates that an expression (usually appearing in a union expression) is redundant, and could be dropped without affecting the value of the enclosing constraint. Examples include expressions such as *(a+b)&c* and constraints 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 rightmost columns of both *a* and *b*.

B.6.2 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 known 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 meaning, but is guaranteed to resolve the reference, since only the *f* declared in *S* will produce a nonempty result. Note that this is *not* a special casting syntax. It relies only on the standard semantics of the domain restriction operator.

B.6.3 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 empty union (that is, 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 for each expression a type that approximates 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 upward or downward. 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: in the former case if *sup* is not in *S*, and in the latter if *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.

B.6.4 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. Functions and predicates, like fields, may be overloaded, so long as all usages can be unambiguously resolved by the type checker.

The constraints implicit in the declarations of arguments of functions and predicates are conjoined to the body constraint when a function or predicate is run. When a function or predicate is invoked (that is,

used within another function or predicate but not run directly), 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.

B.6.5 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 constraints: the constraint *one x: S | F*, for example, says that there is exactly one *x* that satisfies the constraint *F*;
- as quantifiers in quantified expressions: the constraint *lone e*, for example, says that the expression *e* denotes a relation 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 non-empty set of elements drawn from *S*;
- 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, so

```
x: set X
```

would allow *x* to contain any number of elements.

B.7 Language Features

B.7.1 Module Structure

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 an optional header identifying the module, some imports, and some paragraphs:

```
alloyModule ::= [moduleDecl] 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.

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

```
moduleDecl ::= module qualName [[name,+]]
qualName ::= [this/] (name /)* name
```

The path must correspond to the directory location of the module's file with respect to the default root directory, which is the directory of the main file being analyzed. There is a separate default root directory for library models. A module with the module identifier *m* must be stored in the file named *m.als*.

Other modules whose components (signatures, paragraphs or commands) are referred to in this module must be imported explicitly with an import statement:

```
import ::= open qualName [[qualName,+]] [as name]
```

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

Each imported module is referred to within the importing module either by its alias, if given, or if not, by its module identifier. The purpose of aliases is to allow distinct names to be given to modules that happen to share the same module identifier. This arises most commonly when there are multiple imports for the same module with different

parameter instantiations. Since the instantiating types are not part of the module identifier, aliases are used to distinguish the instantiations.

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 signature or with *Int* is an error.

A single module may be imported more than once with the same parameters. The result is *not* to create multiple copies of the same module, but rather to make a single module available under different names. The order of import statements is also immaterial, even if one provides instantiating parameters to another.

If the name of a component of an imported module is unambiguous, it may be referred to without qualification. Otherwise, a *qualified name* must be used, consisting of the module identifier or alias, a slash mark, and then the component name. If an alias is declared, the regular module name may not be used. Note also that qualified names may not include instantiated parameters, so that, as mentioned above, if a single module is imported multiple times (with different instantiating parameters), aliases should be declared and components of the instantiated modules referred to with qualified names that use the aliases as prefixes.

The paragraphs of a module are signatures, facts, predicate and function declarations, assertions, and commands:

```
paragraph ::= sigDecl | factDecl | predDecl | funDecl
   | assertDecl | cmdDecl
```

Paragraphs may appear in a module in any order. There is no requirement of definition before use.

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. Assertions are redundant constraints that are intended to hold, and are checked to ensure consistency. Commands are used to instruct the analyzer to perform model-finding analyses. A module exports as components all paragraphs except for commands.

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

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

B.7.2 Signature Declarations

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 ::= [abstract] [mult] sig name,+ [sigExt] sigBody
mult ::= lone | some | one
sigExt ::= in qualName [+ qualName]*
```

A signature declared with the *extends* keyword is a *type signature*:

```
sigExt ::= extends qualName
```

A type signature introduces a type or subtype. 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 or subtype signature represents a set of elements that is a subset of the union of its *parents*: the signatures listed in its declaration. A subset signature may not be extended, and subset signatures 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 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 the signatures that extend 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 the default overriding keyword *set*) 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 block containing a *signature fact* constraining the elements of the signature:

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

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 field 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 @ suppresses 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 of *C* called *A* and *B*, each with a field *f*.

B.7.3 Declarations

The same declaration syntax is used for fields of signatures, arguments to functions and predicates, comprehension variables, and quantified variables, all of which we shall here refer to generically 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 ::= *[***disj***]* name,$^+$ **:** *[***disj***]* expr

A declaration has two effects:

- Semantically, it constrains the *value* a variable can take. The relation denoted by each variable (on the left) is constrained to be a subset of the relation denoted by the bounding expression (on the right).
- For the purpose of type checking, a declaration gives the variable a *type*. A type is determined for the bounding expression, and that type is assigned to the variable.

When more than one variable is declared at once, the keyword *disj* appearing on the left indicates that the declared variables are mutually disjoint (that is, the relations they denote have empty intersections). In the declarations of fields (within signatures), the *disj* keyword may appear also on the right, for example:

```
sig S {f: disj e}
```

This constrains the field values of distinct members of the signature to be disjoint. In this case, it is equivalent to the constraint

```
all a, b: S | a != b implies no a.f & b.f
```

which can be written more succinctly (using the *disj* keyword in a different role) as

```
all disj a, b: S | disj [a.f, b.f]
```

Any variable that appears in a bounding 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 earlier in the argument declarations. This ordering applies only to variables and not to a signature name, which can appear in a

bounding expression irrespective of where the signature itself is declared.

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 (see section B.7.2). A similar treatment applies to multiplicity constraints (see sections B.6.5 and B.7.4) and *disj*. 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 flavor. 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*.

B.7.4 Multiplicities

In either a declaration

decl ::= *[***disj***]* name,$^+$ **:** *[***disj***]* expr

or a subset constraint

expr ::= expr **in** expr

the right-hand side expression may include *multiplicities*, and the keyword *set* that represents the omission of a multiplicity constraint.

There are two cases to consider, according to whether the right-hand expression denotes a unary relation (ie, a set), or a relation of higher arity.

If the right-hand expression denotes a unary relation, a multiplicity keyword may appear as a unary operator on the expression:

expr ::= unOp expr
unOp ::= mult | **set**
mult ::= **lone** | **some** | **one**

as in, for example:

```
x: lone S
```

which would constrain *x* to be an option—either empty or a scalar in the set *S*.

The multiplicity keywords apply cardinality constraints to the left-hand variable or expression: *lone* says the set contains at most one element; *some* says the set contains at least one element; and *one* says the set contains exactly one element. In a declaration (formed with the colon rather than the *in* keyword), the default multiplicity is *one*, so that the declared variable or expression is constrained to be a singleton as if it were marked with the keyword *one*.

Thus

```
x: S
```

makes *x* a scalar when *S* is a set. The *set* keyword retracts this implicit constraint and allows any number of elements.

If the right-hand expression denotes a binary or higher-arity relation, multiplicity keywords may appear on either side of an arrow operator:

```
expr ::= expr arrowOp expr
arrowOp ::= [mult|set] -> [mult|set]
```

If the right-hand expression has the form *e1 m->n e2*, where *m* and *n* are multiplicity keywords, the declaration or formula imposes a *multiplicity constraint* on the left-hand variable or expression. 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*.

When the expressions *e1* and *e2* are unary, these reduce to familiar notions. For example, the declaration

```
r: X -> one Y
```

makes *r* a total function from *X* to *Y*;

```
r: X -> lone Y
```

makes it a partial function; and

```
r: X one -> one Y
```

makes it a bijection.

Multiplicity markings can be used in nested arrow expressions. For example, a declaration of the form

```
r: e1 m -> n (e2 m' -> n' e3)
```

produces the constraints described above (due to the multiplicity keywords *m* and *n*), but it produces additional constraints (due to *m'* and *n'*). The constraints for the nested expression are the same multiplicity constraints as for a top-level arrow expression, but applied to each image of a tuple under the declared relation that produces a value for the nested expression. For example, if *e1* denotes a set, the multiplicity markings *m'* and *n'* are equivalent to the constraint

```
all x: e1 | x.r in e2 m' -> n' e3
```

If *e1* is not a set, the quantification must range over the appropriate tuples. For example, if *e1* is binary, the multiplicities are short for

```
all x, y: univ | x->y in e1 implies y.(x.r) in e2 m' -> n' e3
```

A subset constraint that includes multiplicities is sometimes called a *declaration formula* (to distinguish it from a *declaration constraint* implicit in a declaration). Declaration formula are useful for two reasons. First, they allow multiplicity constraints to be placed on arbitrary expressions. Thus,

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

says that the join of *p* and *q* is a bijection. Second, they allow additional multiplicity constraints to be expressed for fields that cannot be expressed in their 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's multiplicity applies to the relation *this.r*, it cannot constrain the left-hand multiplicity of the relation. To say that *r* maps at most one *A* to each *B*, one could add as a fact the declaration formula

```
r in A lone->B
```

B.7.5 Expression Paragraphs

A *fact* is a constraint that always holds; from a modeling perspective, it can be regarded as an assumption. A *predicate* is a template for a constraint that can be instantiated in different contexts; one would use predicates, for example, to check that one constraint implies another. 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. (Recall that the grammar unifies constraints and expressions into a single expression class; the terms 'constraint' and 'expression' are used to refer to boolean- and relation-valued expressions respectively.)

A fact can be named for documentation purposes. An assertion can be named or anonymous, but since a command to check an assertion 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 block (which is a sequence of constraints, implicitly conjoined):

factDecl ::= **fact** *[*name*]* block

A predicate declaration consists of the name of the predicate, some argument declarations, and a block of constraints:

predDecl ::= **pred** *[*qualName **.***]* name *[*paraDecls*]* block

paraDecls ::= **(** decl,* **)** | **[** decl,* **]**

(In functions and predicates, either round or square parentheses may be used to delineate the argument list.)

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

pred S.f (…) {…}

the first argument is taken to be a scalar drawn from the 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 function, some argument declarations, and an expression:

funDecl ::= **fun** *[*qualName **.***]* name *[*paraDecls*]* **:** expr **{** expr **}**

```
paraDecls ::= ( decl,* ) | [ decl,* ]
```

The argument declarations include a bounding 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.

Predicates and functions are invoked by providing an expression for each argument; the resulting expression is a boolean expression for a predicate and an expression of the function's return type for a function:

```
expr ::= expr [ expr,* ]
```

In contrast to the declaration syntax, invocations may use only square and not round parentheses; this is a change from a previous version of Alloy. A function instantiation of the form *f[x]* looks just like a primitive function application, where *f* is a relation that is functional and *x* is a set or scalar. Note, however, that this syntactic similarity is only a pun semantically, since instantiation of a declared function is not a relational join: it may be higher order (in the relational sense, mapping relations to relations), and does not have the lifting semantics of a join (namely that application to a set results in the union of application to the set's elements). A predicate application is treated in the same way as a function application, but yields an expression of boolean type: that is, a constraint.

The syntactic similarity is systematic however. An argument list inside the box can be traded for individual boxes, so that *f[a,b]*, for example, can be written equivalently as *f[a][b]*. Likewise, the dot operator can be used in place of the box operator:

```
expr ::= expr binOp expr
binOp ::= .
```

so *f[a]* can be written *a.f*, and any combination of dot and box is permitted, following the rule that the order of arguments declared in the function corresponds to the order of columns in the 'relation' being joined. Finally, the same resolving that allows field names to be overloaded applies to function and predicate names.

There are two predefined functions and predicates: *sum* and *disj*. The *sum* function is discussed below in section B.9. The *disj* predicate returns true or false depending on whether its arguments represent mutually disjoint relations. Unlike a user-defined predicate or function, *disj* accepts any number of arguments (greater than zero). For example, the expression

```
disj [A, B, C]
```

evaluates to true when the sets *A*, *B* and *C* are all mutually disjoint.

Invocation can be viewed as textual inlining. An invocation of a predicate gives a constraint which is obtained by taking the constraint 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. Note that these rules just represent a special case of the equivalence of the dot and box operators; all one must remember is that an argument declared in receiver style is treated as the first argument in the list.

B.7.6 Commands

A command is an instruction to the Alloy Analyzer to perform an analysis. Analysis involves constraint solving: finding an *instance* that satisfies a constraint. A *run* command causes the analyzer to search for an *example* 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 run command consists of an optional command name, the keyword *run*, the name of a function or predicate (or just a constraint given as a block), and, optionally, a scope specification:

```
cmdDecl ::= [name :] run [qualName|block] [scope]
scope ::= for number [but typescope,+] | for typescope,+
typescope ::= [exactly] number qualName
```

A command to check an assertion has the same structure, but uses the keyword *check* in place of *run*, and either the name of an assertion, or a constraint to be treated as an anonymous assertion:

```
cmdDecl ::= [name :] check [qualName|block] [scope]
```

The command name is used in the user interface of the Alloy Analyzer to make it easier to select the command to be executed: when the command name is present, it is displayed instead of the command string itself.

As explained in section B.5, analysis always involves solving a constraint. For a predicate with body constraint *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. Running 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 only when that predicate or function is run directly, but are ignored when the predicate or function is invoked in another predicate or function.

For an assertion whose body constraint 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. That is, checking an assertion yields counterexamples that represent cases in which the facts and declarations hold, but the assertion does not.

An instance 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, the first of which will be named *this* if declared in receiver position without an argument name;
- for an instance of a 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 [but typescope,+] | for typescope,+
typescope ::= [exactly] number qualName
```

The *scope specification* 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 given bounds in a scope specification (although of course any set can be bounded with an explicit constraint on its cardinality). For the rest of this section, "signature" should be read as synonymous with "type signature."

For the built-in signature *Int*, the scope specification does not give the number of elements in the signature. Instead, it gives the bitwidth of integers, including the sign bit; all integers expressible in this bitwidth are included implicitly in the type *Int*. For example, if the scope specification assigns 4 to *Int*, there are four bits for every integer and integer-valued expression, and the set *Int* may contain values ranging from –8 to +7, including zero. All integer computations are performed within the given bitwidth, and if, for a given instance, an expression's evaluation would require a larger bitwidth to succeed without overflow, the instance will not be considered by the analysis.

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 may be associated with any signature, implicitly, explicitly, or by default; and
- *complete*: every top-level signature must have a bound, implicitly or explicitly.
- *uniform*: if a subsignature is explicitly bounded, its ancestor top-level signature must be also.

B.7.7 Expressions

Expressions in Alloy fall into three categories, which are determined not by the grammar but by type checking: *relational expressions*, *boolean expressions*, and *integer expressions*. The term 'expression' without qualification means a relational or integer expression; the term 'constraint' or 'formula' refers to a boolean expression. The category of an expression is determined by its context; for example, the body of a fact, predicate or assertion is always a constraint, and the body of a function is always an expression.

Most operators apply only to one category of expression—the logical operators apply only to constraints, and the relational and arithmetic operators apply only to expressions—with the exception of the conditional construct, expressed with *implies* (or =>) and *else*, and the *let* syntax, which apply to all expression types.

Predicate invocation and function invocation both use the dot and box operators (as explained in section B.7.5), but are treated as constraints and expressions respectively.

A *conditional expression* takes the form

expr ::= expr *(***=>** | **implies***)* expr **else** expr

In the expression

b **implies** e1 **else** e2

b must be a boolean expression, and the result is the value of the expression *e1* when *b* evaluates to true and the value of *e2* when *b* evaluates to false. The expressions *e1* and *e2* may both be boolean expressions, or integer expressions or relational expressions. When they are boolean expressions, the *else* clause may be omitted, in which case *implies* is treated as a simple binary operator, as if *e2* were replaced by an expression that always evaluated to *true* (see section B.10). The keyword *implies* and the symbol => are interchangeable.

A *let expression* allows a variable to be introduced, to highlight an important subexpression or make an expression or constraint shorter by factoring out a repeated subexpression:

expr ::= **let** letDecl,$^+$ blockOrBar
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 the bounding expressions must have been previously declared. Recursive bindings are not permitted.

Any expression may be surrounded by parentheses to force a particular order of evaluation:

```
expr ::= ( expr )
```

B.8 Relational Expressions

An expression may be a constant:

```
expr ::= const
const ::= 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 constraint such as

```
iden in r
```

will be unsatisfiable unless the relation *r* has type *univ*->*univ*. To say that a relation *r* is reflexive with respect to a particular domain type *t*, one might write

```
t <: iden in r
```

An expression may consist of a qualified name, or a simple name prefixed with the special marking @, or the keyword *this*:

```
expr ::= qualName | @name | this
qualName ::= [this/] (name /)* name
```

If the name is the name of a field, 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 bounding expressions and signature facts—the prefix @ preempts dereferencing (see subsection B.7.2). If there is more than one field of the given name, the reference is resolved, or rejected if ambiguous (see section B.6). If the name denotes a quantified or let-bound variable, or the argument of a function or predicate, its value is determined by the binding.

A qualified name can include a path prefix that identifies the module (section B.7.1). In this case, the name may refer to a signature. It may also refer to a predicate or function, if the expression is part of an invocation. A predicate or function without arguments can be invoked either with an empty argument list, or without an argument list at all (section B.7.5). Field names cannot be qualified; to disambiguate a field name, one can write *S* **<:** *f*, for example, to denote the field *f* of signature *S*, and the signature *S* may be given by a qualified name.

Within a predicate or function body, the special keyword *this* refers to an argument declared in receiver position; in a signature fact, it refers to the implicitly quantified member of the signature (see section B.7.2).

Compound expressions may be formed using unary and binary operators in various forms. In this section, we will consider the relational operators alone; as noted above, the grammar does not distinguish expression types, although the type checker does. The expression forms are:

```
expr ::= unOp expr | expr binOp expr | expr arrowOp expr
   | expr [ expr,* ]
unOp ::= ~ | * | ^
binOp ::= & | + | - | ++ | <: | :> | .
arrowOp ::= [mult|set] -> [mult|set]
```

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 box join operators have different precedence, so *a.b[c]* is parsed as *(a.b)[c]*. The dot and box operators are also used for predicate and function invocation, as explained in section B.7.5.

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 presence of multiplicity markings on either side of the arrow adds a constraint, as explained in section B.7.4.

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 B.6).

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*. Note that override *is* defined for relations of arity greater than two, and that in this case, the override is determined using only the first columns.

An expression may be a *comprehension expression*:

expr ::= **{** decl,$^+$ blockOrBar **}**

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*, and so on, and for which the constraint *F* holds. The expressions *e1*, *e2*, and so on, must be unary, and may not be prefixed by (or contain) multiplicity keywords. More general declaration forms are not permitted, except for the use of the *disj* keyword on the left-hand side of the declarations.

B.9 Integer Expressions

Relational expressions are "closed" with respect to the scope, which means that, for any given analysis, if some relations can be represented, then any combination of those relations can be expressed too. This is not true for the integer expressions. As explained in section B.7.6, in an analysis the bitwidth of integers is bounded. Thus while it may be possible to represent two integers, it may not be possible to represent their sum. This means that, when a constraint is analyzed, an instance will not be considered if any expression within the constraint would evaluate (in that instance) to an integer beyond the scope.

An integer expression may be a constant:

```
expr ::= const
const ::= [-] number
```

A numeric literal formed according to the lexical rules (section B.1) represents the number given, as a primitive integer; the analyzer will complain if the number is not expressible within the bitwidth of the analysis scope. The literal may be preceded by a minus sign to denote the negative integer.

Since a signature name may be appear as a constant expression, the built-in signature *Int* may be used to represent the set of integers within scope (as explained in section B.7.6).

There is one unary arithmetic operator, namely cardinality:

```
expr ::= unOp expr
unOp ::= #
```

The expression *#e* denotes (as a primitive integer) the number of tuples in the relation denoted by *e*; for a set (ie, a unary relation), this corresponds simply to the number of elements in the set.

The built-in function *sum* takes a set of integers and returns their sum, as if it were declared as

fun sum (s: **set Int**) : **Int** {...}

Conventional arithmetic expressions are constructed with the following built-in functions, using the standard syntax for function application (that is, using the box or dot operator):

- *plus [a,b]*: returns the sum of *a* and *b*;
- *minus [a,b]*: returns the difference between *a* and *b*;
- *mul [a,b]*: returns the product of *a* and *b*;
- *div [a,b]*: returns the number of times *b* divides into *a*;
- *rem [a,b]*: returns the remainder when *a* is divided by *b*.

The *sum* function is applied implicitly to all the arguments of these functions.

The *sum* quantifier allows a distributed summation to be expressed:

expr ::= quant decl,$^+$ bar expr
quant ::= **sum**

The expression

sum x1: e1, x2: e2, ... | e

denotes the integer sum of the values obtained by evaluating the integer expression *e* by binding *x1*, *x2*, etc to the elements of the sets denoted by *e1*, *e2*, etc, in all possible combinations. The bounding expressions *e1*, *e2*, etc, must denote unary relations.

B.10 Boolean Expressions

There are no built-in boolean constants in Alloy, and boolean values cannot be stored within the tuples of a relation. So a function invocation, whose body is a relational expression, never returns a boolean value. A predicate, in contrast, is a parameterized boolean expression, and its invocation (see section B.7.5) returns a boolean value.

Elementary boolean expressions are formed by comparing two relational or integer expressions using comparison operators:

expr ::= expr *[***!** | **not***]* compare0p expr
compare0p ::= **in** | **=** | **<** | **>** | **=<** | **>=**

The relational comparison operators are defined as follows:

- The expression *e1* **in** *e2* is true when the relation that *e1* evaluates to is a subset of the relation that *e2* evaluates to.

- The expression *e1* = *e2* is true when the relation that *e1* evaluates to the same relation as *e2*, that is *e1* **in** *e2* and *e2* **in** *e1*.

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

As explained in section B.7.3, a boolean expression formed with the *in* keyword may, like a declaration, use multiplicity symbols to impose an additional constraint.

The arithmetic comparison operators are defined as follows:

- The constraint *i* < *j* is true when *i* is less than *j*.
- The constraint *i* > *j* is true when *i* is greater than *j*.
- The constraint *i* =< *j* is true when *i* is less than or equal to *j*.
- The constraint *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. For all these operators, the *sum* function is applied implicitly to their arguments, so that if a non-scalar set of integers is presented, the comparison acts on the sum of its elements.

Note that the equals symbol is *not* overloaded, and continues to have its relational meaning when applied to expressions denoting sets of integers. This means that if *S* and *T* are sets of integer atoms, the expression

```
S = T
```

says that *S* and *T* contain the same set of integers, and consequently, may be false even if both *S* =< *T* and *S* >= *T* are true (when one of the arguments, for example, evaluates to a set of integers containing more than one element).

A constraint in which the comparison operator is negated,

```
e1 not op e2
```

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

```
not e1 op e2
```

Boolean expressions can be combined with operators representing the standard logical connectives:

```
expr ::= un0p expr | expr bin0p expr | expr arrow0p expr
un0p ::= ! | not
bin0p ::= || | or | && | and | <=> | iff | => | implies
```

The constraint **not** *F* is true when the constraint *F* is false, and vice versa. The negation operators **not** and **!** are interchangeable in all contexts.

The meaning of the binary operators is as follows:

- The expression *F* **and** *G* is true when *F* is true and *G* is true.
- The expression *F* **or** *G* is true when one or both of *F* and *G* are true.
- The expression *F* **iff** *G* is true when *F* and *G* are both false or both true.
- The expression *F* **implies** *G* is true when *F* is false or *G* is true.
- The expression *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 connectives may be written interchangeably as symbols: **&&** for **and**, **||** for **or**, **=>** for **implies** and **<=>** for **iff**.

A *block* is a sequence of constraints enclosed in braces:

```
expr ::= block
block ::= { expr* }
```

The constraint

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

is equivalent to the conjunction

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

If the sequence is empty, its meaning is true.

A *quantified expression* is formed by prefixing a relational expression with a multiplicity keyword or the quantifier *no*:

```
expr = unOp expr
unOp ::= no | mult
mult ::= lone | some | one
```

The meaning of such an expression is that the relation contains a count of tuples according to the keyword:

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

A *quantified formula* takes this form:

```
expr ::= quant decl,+ block0rBar
block ::= { expr* }
block0rBar ::= block | bar expr
bar ::= |
```

The expression in the body must be boolean (that is, a constraint and not a relational or arithmetic expression).

It makes no difference whether the constraint body is a single constraint preceded by a vertical bar, or a constraint sequence. The two forms are provided so that the vertical bar can be omitted when the body is a sequence of constraints. 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 braces even when the constraint sequence consists of only a single constraint:

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

These forms are all acceptable and are interchangeable.

The meaning of a quantified formula depends on the quantifier:

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

The range and type of the bound variable is determined by its declaration (see subsection B.7.3). In a sequence of declarations, each declared variable may be bounded by the declarations or previously declared variables. For example, in the expression

```
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 expression with multiple declarations may be regarded, for some quantifiers, as a shorthand for nested expressions, each with a single declaration, this is not true in general. 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 expression may be higher-order: that is, it may bind non-scalar values to variables. Whether the expression is analyzable will depend on whether it can be Skolemized by the analyzer, and, if not, how large the scope is.