SWEN-220
Mathematical Models of Software

Introduction to Alloy
Signatures, Fields, Facts, Predicates, and Assertions
Topics

• What is Alloy?
• Atoms, Sets, Relations
• Signatures, Relations, Fields, Multiplicity
  • Facts and Predicates
• More on Signatures: Subsets & Abstract
• Assertions
• Initial Quick References
Alloy

• The Alloy community web site is at alloy.mit.edu
• Alloy is a language for describing structures and a tool for exploring them. It has been used in a wide range of applications from finding holes in security mechanisms to designing telephone switching networks.
• Alloy allows us to build “declarative” models, as opposed to “operational” models.
  – An operational modeler asks "how would I make X happen?".
  – A declarative modeler asks "how would I recognize that X has happened?".
Alloy & Declarative Modeling

Alloy will allow us to model:

"this property always holds for problems up to size X"

or

"this property does not always hold, and here is a counter example".

For example in a Registration System how would you prove:

• A student cannot be enrolled in courses that are scheduled at the same time.

• A teacher cannot both teach and be a student of the same class.
Alloy Components

• Alloy Language – Model Definition
  – Define structure of the model: relations, sets
  – Model navigation: relational operations (i.e. join)
  – Constraints: facts (propositions) & predicates

• Alloy Model Solver (Analysis)
  – SAT (SATISFIABILITY) Solver
  – For our course all we need to know is that Alloy translates the problem to be analyzed into a (usually huge) boolean formula. This formula is handed to a SAT solver, and the solution is translated back by the Alloy.

• More depth on Alloy for the interested student:
  http://alloy.mit.edu/alloy/faq.html
Atoms

• Alloy models are made from *atoms* and *relations*.

• Atoms are the basic modeling object, they are:
  – Indivisible: have no smaller parts
  – Immutable: properties do not change over time
  – Uninterpreted: do not have any built in property
  – Comparable: For equality (= and !=)
Signatures in Alloy

sig Book{ }
sig Author{ }  
  – Introduces two sets of atomic elements (possibly empty)
  – The sets are disjoint (no common elements)
    • no Book & Author (or Book & Author = none)
  – Both sets are subsets of the universal set \textit{univ}.
    • Book \textit{in} univ // note: \textit{in} is subset (\(\subseteq\))
    • Author \textit{in} univ
  – Top level signatures \textit{partition} univ.
Relations in Alloy

• Atoms are associated with one another using *relations*

• Each relation is a *set of tuples*, where each tuple is an ordered list of atoms.

• There are *no attribute names* as in an RDB.

• Columns of a tuple are selected by *position*. 
Fields are Relations

• Books have Authors
• *wrote* is a relation between Author and Book

\[
\text{sig Book} \{ \}
\text{sig Author} \{
\quad \text{wrote} : \text{set Book}
\}
\]
Fields

• Books have Authors
• *wrote* is a relation between Author and Book

\[
\text{sig Book} \; \{ \}
\text{sig Author} \; \{ \\
\quad \text{wrote} : \text{set Book} \\
\}
\]
Fields

• Books have Authors
• \textit{wrote} is a relation between Author and Book

\begin{verbatim}
sig Book {}
sig Author {
    wrote : set Book
}
\end{verbatim}
Multiplicities

• Multiplicities constrain the size of sets or relations.

• We have 4 multiplicities
  – **one**: exactly one (default if omitted)
    (a Book has one Publisher).
  – **set**: zero or more
    (an Patron checks out zero or more Books)
  – **some**: one or more
    (a Book has one or more Authors)
  – **lone**: zero or one (optional)
    (a Book Copy is loaned to 0 or 1 Patrons)
Multiplicities

• Putting a multiplicity keyword before the signature declaration constrains the number of atoms in the set
  
  ```
  one sig President {} // exactly one President
  some sig Fruit{} // 1 or more Fruits
  ```

• Putting a multiplicity keyword before a field constrains the number of atoms participating in the field's relation
  
  ```
  sig Book {
    publisher : one Publisher, // 1 Publisher (or - publisher : Publisher)
    dict : lone Glossary, // 0 or 1 glossary
    writers : some Author , // 1 or more authors
    reviewers : set Reviewer // 0 or more reviewers
  }
  ```
Lending Library Example
(relationships not shown in ERD)
Lending Library Example
(relationships not shown in ERD)

sig Publisher {
    published : some Book
}

dig Book {
    publisher : one Publisher, 
    writers : some Author, 
    copies : some Copy
}

dig Copy {
    book : Book, 
    loanedTo : lone Patron
}

dig Author {
    wrote : some Book
}

dig Patron {
    loans : set Copy
}
Lending Library Example
(relationships not shown in ERD)

`sig Publisher {`
  `published : some Book`
`}`

`sig Book {`
  `publisher : one Publisher,`
  `writers : some Author,`
  `copies : some Copy`
`}`

`sig Copy {`
  `book : Book,`
  `loanedTo : one Patron`
`}`

`sig Author {`
  `wrote : some Book`
`}`

`sig Patron {`
  `loans : set Copy`
`}`
Lending Library Example
(relationships not shown in ERD)

```
sig Publisher {
    published : some Book
}

sig Book {
    publisher : one Publisher,
    writers : some Author,
    copies : some Copy
}

sig Copy {
    book : Book,
    loanedTo : lone Patron
}

sig Author {
    wrote : some Book
}

sig Patron {
    loans : set Copy
}
```

We'll have to "connect" each pair to ensure consistency.
Facts

- Facts define the "rules of the game."
- A fact is a proposition (true or false)
- A fact restricts "legal" solutions to cases where the fact is true.
- Let's define a consistency fact for our library relations:

```
fact LibraryConsistency {
    publisher = ~published
    writers = ~wrote
    book = ~copies
    loanedTo = ~loans
}
```

(*) If rel is a binary relation (tuples = pairs) then ~rel is the relational inverse, which is the result of swapping the elements in each pair.
Facts

- Facts define the "rules of the game."
- A fact is a proposition (true or false)
- A fact restricts "legal" solutions to cases where the fact is true.
- Let's define a consistency fact for our library relations:

```java
fact LibraryConsistency {
    publisher = ~published
    writers = ~wrote
    book = ~copies
    loanedTo = ~loans
}
```

(\*) If $\text{rel}$ is a binary relation (tuples = pairs) then $\sim\text{rel}$ is the relational inverse, which is the result of swapping the elements in each pair.

Propositions written on separate lines are conjoined ("anded") together.
**Facts**

- Facts define the "rules of the game."
- A fact is a proposition (true or false)
- A fact restricts "legal" solutions to cases where the fact is true.
- Let's define a consistency fact for our library relations:

```plaintext
fact LibraryConsistency {
    publisher = ~published and
    writers = ~wrote and
    book = ~copies and
    loanedTo = ~loans
}
```

(*) If `rel` is a binary relation (tuples = pairs) then `~rel` is the *relational inverse*, which is the result of swapping the elements in each pair.
Digression on ~

• Alloy modelers rarely include both pairs of inverse relations like publisher/published, writers/wrote, book/copies, and loanedTo/loans.
• Most will include just one of the pair (e.g., the most used).
• If they need the other relation, they'll just apply ~.

```
sig Book {
    publisher : one Publisher,
    writers : some Author,
}
sig Author {
}
sig Patron {
}
sig Copy {
    book : Book
    loanedTo : lone Patron
}
sig Publisher {
}
```

If \( c \) is a Copy and \( p \) is a Patron, then

\[ c.\text{loanedTo} \] is the Patron who has the copy \( c \) (if any).
\[ p.\sim\text{loanedTo} \] is the set of all the Copies on loan to \( p \).
The Model Solution Space

All possible solutions (no facts)

Instances created by Alloy when the model is executed - “run”

Selecting “next” shows additional solutions
**The Model Solution Space**

Instances created by Alloy when the model is executed – “run”

Selecting “next” shows additional solutions

*Valid Instances

Invalid Instances

All possible solutions (no facts)

*Valid = Our perception of what a valid instance should be. Alloy will create all possible instances based on our definition of the model (signatures & facts) despite our perception of validity.
The Model Solution Space

All possible solutions when **Facts** (propositions) are applied

*Valid* = Our perception of what a valid instance should be. Alloy will create all possible instances based on our definition of the model (signatures & facts) despite our perception of validity.
The Model Solution Space

All possible solutions when **Facts** (propositions) and **Predicates** are applied

**Predicates** constrain the solution space to a specific partition of valid instances

*Valid* = Our perception of what a valid instance should be. Alloy will create all possible instances based on our definition of the model (signatures & facts) despite our perception of validity.
Predicates

• Predicates (*preds*) add constraints.
• Used to explore the state space of legal solutions according to the facts:

```plaintext
pred showAll{ }  
run showAll     // same as run{}

pred someLoans {  
  some loans // at least 1 each of Patron, Copy, Book, Publisher & Author  
}  
run someLoans

pred onePublisherWithLoans { // all books from a single publishing source  
  one Publisher  
  someLoans     // note the reference to predicate someLoans  
}  
run onePublisherWithLoans
```
Signature Subsets
-Extensions

sig Book {}
sig Novel extends Book {}
sig History extends Book {}
sig Biography extends Book {}
sig Author {}

• Top level signatures and extensions of the same signature are disjoint
Abstract Signatures

abstract sig Book {}

sig Novel extends Book {}

sig History extends Book {}

sig Biography extends Book {}

- An *abstract signature* has no atoms on its own.
- All of its atoms belong to its extensions.
- That is, the extensions *partition* the abstract signature.
- Analogous to abstract classes in Java.
Signature Subsets: in

abstract sig Book {}
sig Novel extends Book {}
sig History extends Book {}
sig Biography extends Book {}
sig Humorous in Book {}

- Humorous, in any solution, is a subset of Book.
- It may contain some of the Novels, some of the Histories, and some of the Biographies.
- In any solution it is "fixed".
Example – Love & Marriage

abstract sig Person {}
sig Male extends Person {}
sig Female extends Person {}
sig Married in Person {}

Lets add a first name field (fname) for each Person:

sig Name{}
abstract sig Person {
    fname : one Name
}
sig Male extends Person {}
sig Female extends Person {}
sig Married in Person {}
Example – Love & Marriage

```
sig Name{}
abstract sig Person {
    fname : one Name
}
sig Male extends Person {}
sig Female extends Person {}
sig Married in Person {}
```

How would you include a relation "spouse" for married persons?
Example – Love & Marriage

```
sig Name{}
abstract sig Person {
  fname : one Name
}
sig Male extends Person {}
sig Female extends Person {}
sig Married in Person {
  spouse : Married
}
```

How would you include a relation "spouse" for married persons?
Example – Love & Marriage

\texttt{sig Name{}\}
\texttt{abstract sig Person \{\}
\quad \texttt{fname : one Name} \quad \texttt{// we use one to enforce one name per person\}
\}
\texttt{sig Male extends Person {}\}
\texttt{sig Female extends Person {}\}
\texttt{sig Married in Person \{}
\quad \texttt{spouse : Married} \quad \texttt{// note the default multiplicity is one\}
\}

We’ll add two facts:

1. - Every \texttt{Person} must have a unique \texttt{Name}
2. - Every \texttt{Married Person} must have a \texttt{spouse} of the opposite sex
   ( ...for the purposes of this academic example 😊 )
Example – Love & Marriage

\[
\text{sig Name}\{
\text{abstract sig Person} \{ \\
\text{ \hspace{1cm} fname : one Name} \hspace{1cm} \text{// we use one to enforce one name per person} \\
\} \\
\text{sig Male extends Person} \{}
\text{sig Female extends Person} \{}
\text{sig Married in Person} \{ \\
\text{ \hspace{1cm} spouse : Married} \hspace{1cm} \text{// note the default multiplicity is one} \\
\}
\]

Fact #1 :
1. - Every \text{Person} must have a unique \text{Name}

\[
\text{fact UniqueName}\{ \\
\text{all p1, p2 : Person} \mid \text{p1.fname != p2.fname} \\
\}
\]
Example – Love & Marriage

Fact #1:

- Every Person must have a unique Name

```plaintext
fact UniqueName{
    all p1, p2 : Person | p1.fname != p2.fname
}
```

This is a “quantified logical expression” (see end of slides for a full reference sheet)

```
all decls | predicate
    The predicate is true for all combinations of the variables declared in decls.
    (The universal quantifier)
```

Our fact UniqueName is evaluated (true or false)

“It is true for all combinations of Person (atoms) using variables p1, p2,
that ( | ) the fnames of those two Persons are not the same (!=)”

If the expression evaluates false, no Person atoms will be created as we used the all qualifier.
Fact #1:

- Every Person must have a unique Name

```alloy
fact UniqueName{
    all p1, p2 : Person | p1.fname != p2.fname
}
```

Try executing that and you will be surprised to find no Persons are created in any of the possible solutions!

We must be precise in our model definition. In the above expression we told Alloy to take every (all) possible combinations of People. A valid combination is the pair of same People. We meant to say all combinations of different People, so:

```alloy
fact UniqueName{
    all disjoint p1, p2 : Person | p1.fname != p2.fname  // works better!
}
```
Fact #1:

- Every Person must have a unique Name

```
fact UniqueName{
    all disjoint p1, p2 : Person | p1.fname != p2.fname
}
```

Valid alternative quantified logical expressions:

```
no disjoint p1, p2 : Person | p1.fname = p2.fname
“It is never true (no) that a combination of Person (atoms) using variables p1, p2 that ( | ) the fnames of those two Persons are the same (=)”

all disjoint p1, p2 : Person | no p1.fname & p2.fname
“It is always true (all) that a combination of Person (atoms) using variables p1, p2 that ( | ) the union (&) of the sets p1.fname and p2.fname is empty (no)”
```

The latter, using relational operators, is often the preferred approach as it keeps us away from using equality (=, !=) which can sometimes be troublesome if we are not careful.
Example – Love & Marriage

abstract sig Person {}

sig Male extends Person {}

sig Female extends Person {}

sig Married in Person {
    spouse : Married
}

Fact #2

How might you indicate that the spouses are of different sexes?

fact SpouseSexesDiffer {
    all m : Married | m in Male => m.spouse in Female
}

“It always true that for all Married Persons m that (|) if m is a subset (in) of set Male this implies (=>) that his spouse (m.spouse) is a subset (in) of set Female”

Note: we still have an issue here. Run the model with this fact to see the problem.
The Model Solution Space
(wwhat we think...)

All possible solutions when **Facts**
(propositions) are applied

*Valid* = Our perception of what a valid instance should be. Alloy will create all possible instances based on our definition of the model (signatures & facts) despite our perception of validity.
The Model Solution Space
(what Alloy thinks...)

All possible solutions when **Facts**
(propositions) are applied

Solutions as a result of incomplete / erroneous definition

*Valid = Our perception of what a valid instance should be. Alloy will create all possible instances based on our definition of the model (signatures & facts) despite our perception of validity.
Assertions

• Checking all of the solutions that satisfy our models can be tedious.
• This is where we can use assertions!
  • Assertions are *propositional claims*.
  • They indicate what we believe (or hope) are a consequences of our model & facts.
  • Assertions are checked for *counterexamples*. 
All possible solutions when **Facts** are applied and an **Assertion** is checked for counterexamples

**Counterexamples** created by Alloy when we check assertions
Example – Love & Marriage

abstract sig Person {}
sig Male extends Person {}
sig Female extends Person {}
sig Married in Person {
  spouse : Married
}

fact SpouseSexesDiffer {
  all m : Married | m in Male => m.spouse in Female
}

Asserting spouses are of different sexes:

assert DifferentSexes {
  no m : Married {
    (m + m.spouse) in Male or
    (m + m.spouse) in Female
  }
}

check DifferentSexes
Example – Love & Marriage

abstract sig Person {}
sig Male extends Person {}
sig Female extends Person {}
sig Married in Person {
    spouse : Married
}

fact SpouseSexesDiffer {
    all m : Married | m in Male => m.spouse in Female
}

Asserting spouses are of different sexes:

assert DifferentSexes {
    no m : Married {
        (m + m.spouse) in Male or
        (m + m.spouse) in Female
    }
}

check DifferentSexes

Assertions are checked, not run.
Example – Love & Marriage

abstract sig Person {}

sig Male extends Person {}

sig Female extends Person {}

sig Married in Person {
    spouse : Married
}

fact SpouseSexesDiffer {
    all m : Married | m in Male => m.spouse in Female
}

Asserting spouses are of different sexes:

assert DifferentSexes {
    no m : Married {
        (m + m.spouse) in Male or
        (m + m.spouse) in Female
    }
}

check DifferentSexes

For long predicates we can surround the body in curly braces, and individual claims on different lines are conjoined.
Example – Love & Marriage

abstract sig Person {}
sig Male extends Person {}
sig Female extends Person {}
sig Married in Person {
spouse : Married
}

fact SpouseSexesDiffer {
  all m : Married | m in Male => m.spouse in Female
}

Asserting spouses are of different sexes:

assert DifferentSexes {
  no m : Married {
    (m + m.spouse) in Male or
    (m + m.spouse) in Female
  }
}

check DifferentSexes

Same as
no m : Married | (m + m.spouse) in Male or
(m + m.spouse) in Female
Example – Love & Marriage

abstract sig Person {}
sig Male extends Person {}
sig Female extends Person {}
sig Married in Person {
    spouse : Married
}

fact SpouseSexesDiffer {
    all m : Married | m in Male => m.spouse in Female
}

Asserting spouses are of different sexes:

assert DifferentSexes {
    no m : Married {
        (m + m.spouse) in Male or
        (m + m.spouse) in Female
    }
}

check DifferentSexes

no is the opposite of all.
Here we claim we cannot find a married person with the given property when all the facts hold.
Example – Love & Marriage

abstract sig Person {}

sig Male extends Person {}

sig Female extends Person {}

sig Married in Person {
    spouse : Married
}

fact SpouseSexesDiffer {
    all m : Married | m in Male => m.spouse in Female
}

Asserting spouses are of different sexes:

assert DifferentSexes {
    no m : Married {
        (m + m.spouse) in Male or
        (m + m.spouse) in Female
    }
}

check DifferentSexes

Same as
all m : Married | (m + m.spouse) ! in Male and
(m + m.spouse) ! in Female
Example – Love & Marriage

```
abstract sig Person {}
sig Male extends Person {}
sig Female extends Person {}
sig Married in Person {
    spouse : Married
}

fact SpouseSexesDiffer {
    all m : Married | m in Male => m.spouse in Female
}
```

Asserting spouses are of different sexes:

```
assert DifferentSexes {
    no m : Married {
        (m + m.spouse) in Male or
        (m + m.spouse) in Female
    }
}

check DifferentSexes
```

"+" is set union.

Here the set of a married person and his or her spouse.
Example – Love & Marriage

abstract sig Person {}
sig Male extends Person {}
sig Female extends Person {}
sig Married in Person {
spouse : Married
}

fact SpouseSexesDiffer {
  all m : Married | m in Male => m.spouse in Female
}

Asserting spouses are of different sexes:

assert DifferentSexes {
  no m : Married {
    (m + m.spouse) in Male or
    (m + m.spouse) in Female
  }
}

check DifferentSexes

If \((m + m\text{.spouse})\) in Male then both persons are Male. Same for Female.
The claim is that this never happens.
Does the claim hold? Try it out!

Try rewriting the assertion using all instead of no.
Implies Operator

Assume $B1$ and $B2$ are boolean values (typically the results of a boolean expressions):

$B1$ implies $B2$ (or $B1 => B2$) is true if and only if $B1$ is false or $B2$ is true.

Same as (not $B1$) or $B2$

*Be very careful with implication* - it is commonly read as "if $B1$ then $B2$". The problem is that when $B1$ is false, $B2$ can be *either* true or false.

Consider: If it is raining I will carry an umbrella. (i.e., $R => U$)

We only know what will happen when it is raining.

If it is not raining then I may or may not carry an umbrella!

$B1$'s truth is *sufficient* for $B2$'s truth, but not necessary.

$B2$'s truth is *necessary* for $B1$'s truth, but not sufficient.

<table>
<thead>
<tr>
<th>$B1$</th>
<th>$B2$</th>
<th>$B1 =&gt; B2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$m$ in Male</th>
<th>$m$.spouse in Female</th>
<th>$B1 =&gt; B2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

This is a convention, which can seem counterintuitive at times, but which is very useful. Intuitively the convention encodes the fact that *if $B1$ is false, the implication tells us nothing about $B2$*. 


Equivalence Operator

Assume $B_1$ and $B_2$ are boolean values (typically the results of a boolean expressions):

$B_1$ equiv $B_2$ (or $B_1 \iff B_2$) is true if and only if $B_1$ and $B_2$ have the same truth value (both true or both false).

Same as $(B_1 \Rightarrow B_2)$ and $(B_2 \Rightarrow B_1)$

$B_1$ is necessary and sufficient for $B_2$ (and vice versa). $B_1$ and $B_2$ make identical claims (i.e., say the same thing)

<table>
<thead>
<tr>
<th>B1</th>
<th>B2</th>
<th>B1 &lt;=&gt; B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>m in Male</th>
<th>m.spouse in Female</th>
<th>B1 &lt;=&gt; B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

all $m : \text{Married} \mid m \text{ in Male} \iff m.\text{spouse in Female}$

(same as)

all $m : \text{Married} \mid (m \text{ in Male} \Rightarrow m.\text{spouse in Female}) \text{ and} 
(m \text{ in Female} \Rightarrow m.\text{spouse in Male})$
Example – Love & Marriage

\[\text{abstract \ sig \ Person \{}\]
\[\text{sig \ Male \ extends \ Person \{}\]
\[\text{sig \ Female \ extends \ Person \{}\]
\[\text{sig \ Married \ in \ Person \{}\]
\[\text{spouse : Married }\]
\[\}\]
\[\text{fact \ SpouseSexesDiffer \{}\]
\[\quad \text{all \ m : Married | m in Male} \leftrightarrow \text{m.spouse in Female} \quad /// \text{Corrected!}\]
\[\}\]

Asserting spouses are of different sexes:

\[\text{assert \ DifferentSexes \{}\]
\[\quad \text{no \ m : Married \{}\]
\[\quad \quad (m + m.spouse) \text{ in Male or }\]
\[\quad \quad (m + m.spouse) \text{ in Female}\]
\[\quad \}\]
\[\} \]
\[\text{check \ DifferentSexes}\]
Quick Reference: Set & Relation Operations

Assume $S_1$ and $S_2$ are sets (signatures, sets associated with fields of a signature, relations (which are sets of tuples), or the result of a set operation):

- $S_1 + S_2$ set union (all elements in either set)
- $S_1 \& S_2$ set intersection (all elements common to both sets).
- $S_1 - S_2$ set difference (all elements in $S_1$ that are not in $S_2$)

"." (simple & incomplete definition)
If $e$ is an element in some signature $S$, and if $f$ is a field in $S$, then $e.f$ is the set of elements (possibly empty) associated with field $f$ for element $e$.

"~" (relational inverse)
If $rel$ is a binary relation (such as a simple field of a signature), then $\sim rel$ is the relation formed by swapping the first and second elements in each tuple in $rel$.

NOTE: If $rel = \sim rel$ holds we say $rel$ is "symmetric:"
Quick Reference: Sig & Field Declaration Constraints

Multiplicity constraints can be placed on the fields in a signature using `some`, `one`, `lone`, and `set`:

- **some sig** S{} // signature (set) S has one or more atoms - it cannot be empty.
- **lone sig** T{} // signature (set) T has zero or one atoms.
- **one sig** U{} // signature (set) U has exactly one atom.
- **sig** V{} // signature (set) V has 0 or more atoms - it may be empty.
- **sig** W{} // etc.
- **sig** X{} // etc.
- **sig** Z{} // etc.

Multiplicity constraints may also be placed on fields (relations) in a signature:

**sig** Example {
  f1 : **set** S, // f1 relates each atom in Example to 0 or more atoms in S
  f2 : **some** V, // f2 relates each atom in Example to 1 or more atoms in V
  f3 : **one** W, // f3 relates each atom in Example to exactly 1 atom in W
  f4 : X, // f4 relates each atom in Example to exactly 1 atom in X
  f5 : **lone** Z, // f5 relates each atom in Example to 0 or 1 atoms in Z
}


Quick Reference: Set & Relation Queries

Assume $S$ is a set (a signature, a set associated with fields of a signature, a relation (which is a set of tuples), or the result of a set operation):

- **some** $S$ is true if and only if $S$ contains at least one element.

- **no** $S$ is true if and only if $S$ contains no elements (is empty).
  Most Alloy modelers prefer this to $S = \text{none}$ because, while none is the empty set, it is the empty set of simple elements. You cannot compare a general relation $R$ to none, but you can determine whether the relation is empty via no $R$.

- **one** $S$ is true if and only if $S$ contains exactly one element.

- **lone** $S$ is true if and only if $S$ contains no elements or exactly one element. lone $S$ is equivalent to (no $S$ or one $S$).
Quick Reference: Boolean Expressions

Assume $B1$ and $B2$ are boolean values (typically the results of a boolean expressions):

- $! B1$ is true if and only if $B1$ is false.
- $B1$ and $B2$ (or $B1 \&\& B2$) is true if and only if $B1$ and $B2$ are both true.
- $B1$ or $B2$ (or $B1 \mid\!\mid B2$) is true if and only if at least one of $B1$ or $B2$ is true.
- $B1$ implies $B2$ (or $B1 \Rightarrow B2$) is true if and only if $B1$ is false or $B2$ is true.

Same as $(\text{not } B1)$ or $B2$

Be very careful with implication - it is commonly read as "if $B1$ then $B2$". The problem is that when $B1$ is false, $B2$ can be either true or false.

Consider: If it is raining I will carry an umbrella. (i.e., $R \Rightarrow U$)
- We only know what will happen when it is raining.
- If it is not raining then I may or may not carry an umbrella!

$B1$'s truth is sufficient for $B2$'s truth, but not necessary.
$B2$'s truth is necessary for $B1$'s truth, but not sufficient.

$B1$ equiv $B2$ (or $B1 \iff B2$) is true if and only if $B1$ and $B2$ have the same truth value (both true or both false).
- Same as $(B1 \Rightarrow B2)$ and $(B2 \Rightarrow B1)$
- $B1$ is necessary and sufficient for $B2$ (and vice versa). $B1$ and $B2$ make identical claims (i.e., say the same thing)
Quick Reference: Quantified Logical Expressions

all decls | predicate
The predicate is true for all combinations of the variables declared in decls.
(The universal quantifier)

some decls | predicate
The predicate is true for some (at least one) combinations of the variables declared in decls.
(The existential quantifier)

no decls | predicate
The predicate is true for none of combinations of the variables declared in decls.
Same as ! some decls | predicate
(The non-existence quantifier)

one decls | predicate
The predicate is true for exactly one combination of the variables declared in decls.
(The uniqueness quantifier)

lone decls | predicate
The predicate is true for none or one combination of the variables declared in decls.
(The optional quantifier)

Note: quant decls | predicate can also be written as quant decls { predicate }.
In the latter case, successive lines of syntactically complete sub-predicates are implicitly conjoined.