



---

## Alloy – syntax and semantics

Video available here

<https://web.microsoftstream.com/video/19e80b3f-530b-4cf0-a576-b36f6e493b48>

---

# Alloy: Syntax and Semantics



- Alloy is a language
  - ▶ It has a syntax *how do I write a right specification?*
  - ▶ It has a semantics *what does it mean?*
- In a programming language
  - ▶ Syntax defines correct programs (i.e., allowed programs)
  - ▶ Semantics defines the meaning of a program as its possible (many?) computations
- In Alloy
  - ▶ Syntax as usual...
  - ▶ Semantics defines the meaning of a specification as the collection of its models, i.e., of the worlds that make our Alloy description true



# Alloy = logic + language + analysis

---

- Logic
    - ▶ first order logic + relational calculus
  - Language
    - ▶ syntax for structuring specifications in the logic
  - Analysis
    - ▶ shows bounded snapshots of the world that satisfy the specification
    - ▶ bounded exhaustive search for counterexample to a claimed property using SAT
-



- Relations
- Constants
- Set Operators
- Join Operators
- Quantifiers and Boolean Operators
- Set and Relation Declaration
- If and let

# Logic: relations of atoms

---



- Atoms are Alloy's primitive entities
    - ▶ indivisible, immutable, uninterpreted
  - Relations associate atoms with one another
    - ▶ set of tuples, tuples are sequences of atoms
-



# Logic: everything is a relation

---

- Sets are unary (1 column) relations

$\text{Name} = \{ (N0), (N1), (N2) \}$        $\text{Addr} = \{ (A0), (A1), (A2) \}$        $\text{Book} = \{ (B0), (B1) \}$

- Scalars are singleton sets

$\text{myName} = \{ (N1) \}$   
 $\text{yourName} = \{ (N2) \}$   
 $\text{myBook} = \{ (B0) \}$

- Ternary relation

$\text{addr} = \{ (B0, N0, A0), (B0, N1, A1), (B1, N1, A2), (B1, N2, A2) \}$

---



- Relation = set of ordered n-tuples of atoms
  - ▶ n is called the *arity* of the relation
- Relations in Alloy are typed
  - ▶ Determined by the declaration of the relation
  - ▶ Example: a relation with type  
`Person -> String`  
only contains pairs
    - whose first component is a Person
    - and whose second component is a String

# Logic: constants



none	<i>empty set</i>
univ	<i>universal set</i>
iden	<i>identity relation</i>

Name = { (N0), (N1), (N2) }

Addr = { (A0), (A1) }

none = { }

univ = { (N0), (N1), (N2), (A0), (A1) }

iden = { (N0,N0), (N1,N1), (N2,N2), (A0,A0),  
(A1,A1) }



# Logic: set operators



+	<i>union</i>
&	<i>intersection</i>
-	<i>difference</i>
in	<i>subset</i>
=	<i>equality</i>

```
greg = { (N0) }  
rob  = { (N1) }  
  
greg + rob    = { (N0), (N1) }  
greg = rob    = false  
rob in none   = false
```

```
Name  = { (N0), (N1), (N2) }  
Alias  = { (N1), (N2) }  
Group  = { (N0) }  
RecentlyUsed = { (N0), (N2) }
```

```
Alias + Group = { (N0), (N1), (N2) }  
Alias & RecentlyUsed = { (N2) }  
Name - RecentlyUsed  = { (N1) }  
RecentlyUsed in Alias = false  
RecentlyUsed in Name  = true  
Name = Group + Alias  = true
```

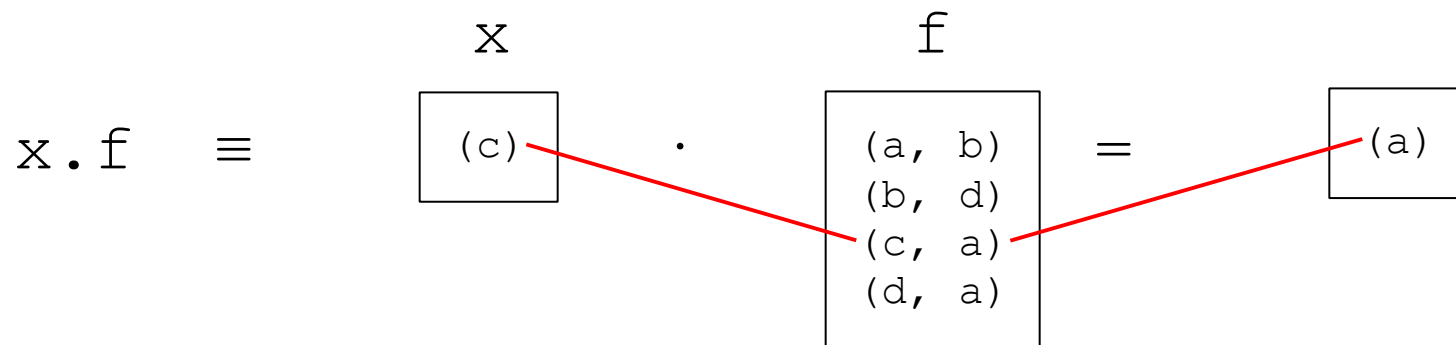
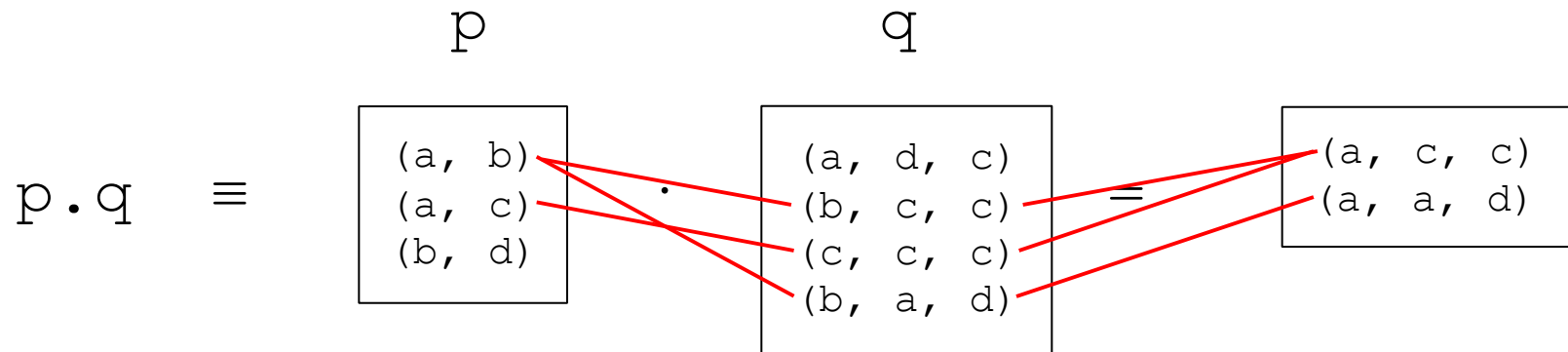
```
cacheAddr = { (N0, A0), (N1, A1) }  
diskAddr  = { (N0, A0), (N1, A2) }
```

```
cacheAddr + diskAddr = { (N0, A0), (N1, A1), (N1, A2) }  
cacheAddr & diskAddr = { (N0, A0) }  
cacheAddr = diskAddr = false
```





# Logic: relational composition dot join



in db join the match of columns is by name, not by position  
and the matching column is not dropped



# Logic: join operators

$\cdot$  *dot join*

$[]$  *box join*

$e1[e2] = e2.e1$   
 $a.b.c[d] = d.(a.b.c)$

```
Book = { (B0) }
```

```
Name = { (N0), (N1), (N2) }
```

```
Addr = { (A0), (A1), (A2) }
```

```
Host = { (H0), (H1) }
```

```
myName = { (N1) }
```

```
myAddr = { (A0) }
```

```
address = { (B0, N0, A0), (B0, N1, A0), (B0, N2, A2) }
```

```
host = { (A0, H0), (A1, H1), (A2, H1) }
```

```
Book.address = { (N0, A0), (N1, A0), (N2, A2) }
```

```
Book.address[myName] = { (A0) }
```

```
Book.address.myName = { }
```

```
host[myAddr] = { (H0) }
```

```
address.host = { (B0, N0, H0), (B0, N1, H0), (B0, N2, H1) }
```



# Logic: unary operators

$\sim$      *transpose*  
 $\wedge$      *transitive closure*  
 $*$      *reflexive transitive closure*

*apply only to binary relations*

$$\begin{aligned}\wedge r &= r + r.r + r.r.r + \dots \\ *r &= \text{iden} + \wedge r\end{aligned}$$

```
Node = { (N0), (N1), (N2), (N3) }  
next = { (N0, N1), (N1, N2), (N2, N3) }     a list  
  
~next = { (N1, N0), (N2, N1), (N3, N2) }  
^next = { (N0, N1), (N0, N2), (N0, N3),  
          (N1, N2), (N1, N3),  
          (N2, N3) }  
*next = { (N0, N0), (N0, N1), (N0, N2), (N0, N3),  
          (N1, N1), (N1, N2), (N1, N3),  
          (N2, N2), (N2, N3), (N3, N3) }
```

```
first = { (N0) }  
rest = { (N1), (N2), (N3) }
```

```
first.^next = rest  
first.*next = Node
```



# Logic: restriction and override

$<:$      *domain restriction*  
 $:>$      *range restriction*  
 $++$      *override*

*Apply to any relations  
(normally binary)*

$p ++ q =$   
 $p - (\text{domain}[q] <: p) + q$

Name     = { (N0), (N1), (N2) }  
Alias     = { (N0), (N1) }  
Addr      = { (A0) }  
address = { (N0, N1), (N1, N2), (N2, A0) }

address  $:>$  Addr = { (N2, A0) }  
Alias  $<:$  address = { (N0, N1), (N1, N2) }  
address  $:>$  Alias = { (N0, N1) }

workAddress = { (N0, N1), (N1, A0) }  
address  $++$  workAddress = { (N0, N1), (N1, A0), (N2, A0) }

$m' = m ++ (k \rightarrow v)$   
*update map  $m$  with key-value pair  $(k, v)$*



# Logic: restriction and override

- Examples of usage of these operators with non binary relations

- ▶  $\text{addr} = \{ (B0, N1, A2), (B1, N1, A2), (B2, N0, A1), (B2, N1, A0) \}$
- ▶  $\{B2\} <: \text{addr} = \{ (B2, N0, A1), (B2, N1, A0) \}$
- ▶  $\text{addr} :> \{A2\} = \{ (B0, N1, A2), (B1, N1, A2) \}$
- ▶  $\text{addr} ++ \{ (B0, N1, A0) \} = \{ (B0, N1, A0), (B1, N1, A2), (B2, N0, A1), (B2, N1, A0) \}$
- ▶  $\text{addr} ++ \{ (B0, N0, A0) \} = \{ (B0, N0, A0), (B1, N1, A2), (B2, N0, A1), (B2, N1, A0) \}$
- ▶  $\text{addr} :> \{N1\} = \{ \}$

# Logic: boolean operators



!	not	<i>negation</i>
&&	and	<i>conjunction</i>
	or	<i>disjunction</i>
=>	implies	<i>implication</i>
,	else	<i>alternative</i>
<=>	iff	<i>bi-implication</i>

*four equivalent constraints:*

`F => G else H`

`F implies G else H`

`(F && G) || ((!F) && H)`

`(F and G) or ((not F) and H)`



# Logic: quantifiers



```
all x: e | F
all x: e1, y: e2 | F
all x, y: e | F
all disj x, y: e | F
```

all	<i>F holds for every x in e</i>
some	<i>F holds for at least one x in e</i>
no	<i>F holds for no x in e</i>
lone	<i>F holds for at most one x in e</i>
one	<i>F holds for exactly one x in e</i>

```
some n: Name, a: Address | a in n.address
```

*some name maps to some address — address book not empty*

```
no n: Name | n in n.^address
```

*no name can be reached by lookups from itself — address book acyclic*

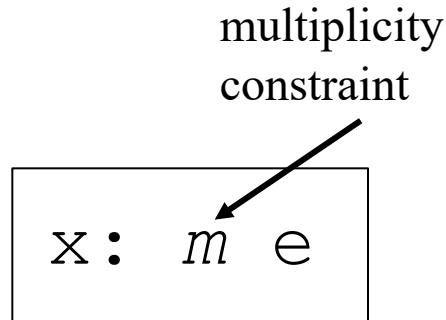
```
all n: Name | lone a: Address | a in n.address
```

*every name maps to at most one address — address book is functional*

```
all n: Name | no disj a, a': Address | (a + a') in n.address
```

*no name maps to two or more distinct addresses — same as above*

# Logic: relation declarations



set	<i>any number</i>
one	<i>exactly one</i>
lone	<i>zero or one</i>
some	<i>one or more</i>

— default

**RecentlyUsed:** set Name

*RecentlyUsed is a subset of the set Name*

**senderAddress:** Addr

*senderAddress is a singleton subset of Addr*

**senderName:** lone Name

*senderName is either empty or a singleton subset of Name*

**receiverAddresses:** some Addr

*receiverAddresses is a nonempty subset of Addr*

# Logic: relation declarations

---


$$r: A \rightarrow B$$
$$(r: A \rightarrow B) \iff ((\forall a: A \mid \exists b: B \mid r(a,b)) \text{ and } (\forall b: B \mid \exists a: A \mid r(a,b)))$$

**workAddress: Name  $\rightarrow$  1one Addr**  
*each name refers to at most one work address*

**homeAddress: Name  $\rightarrow$  one Addr**  
*each name refers to exactly one home address*

**members: Group 1one  $\rightarrow$  some Addr**  
*address belongs to at most one group  
and group contains at least one address*

---

# Logic: set definitions

---


$$\{x1: e1, x2: e2, \dots, xn: en \mid F\}$$
$$\{n: \text{Name} \mid \text{no } n.^{\wedge}\text{address} \ \& \ \text{Addr}\}$$

*set of names that don't resolve to any actual addresses*

$$\{n: \text{Name}, a: \text{Address} \mid n \rightarrow a \text{ in } ^{\wedge}\text{address}\}$$

*binary relation mapping names to reachable addresses*

---

# Logic: if and let



```
f implies e1 else e2
let x = e | formula
let x = e | expression
```

*four equivalent constraints:*

```
all n: Name |
  (some n.workAddress
    implies n.address = n.workAddress
    else n.address = n.homeAddress)
```

```
all n: Name |
  let w = n.workAddress, a = n.address |
    (some w implies a = w else a = n.homeAddress)
```

```
all n: Name |
  let w = n.workAddress |
    n.address = (some w implies w else n.homeAddress)
```

```
all n: Name |
  n.address = (let w = n.workAddress |
    (some w implies w else n.homeAddress))
```

# Logic: cardinalities



$\#r$	<i>number of tuples in <math>r</math></i>
$0, 1, \dots$	<i>integer literal</i>
$+$	<i>plus</i>
$-$	<i>minus</i>

$=$	<i>equals</i>
$<$	<i>less than</i>
$>$	<i>greater than</i>
$=<$	<i>less than or equal to</i>
$>=$	<i>greater than or equal to</i>

$\text{sum } x: e \mid ie$

*sum of integer expression  $ie$  for all singletons  $x$  drawn from  $e$*

$\text{all } b: \text{Bag} \mid \#b.\text{marbles} \leq 3$   
*all bags have 3 or less marbles*

$\# \text{Marble} = \text{sum } b: \text{Bag} \mid \#b.\text{marbles}$   
*the sum of the marbles across all bags  
equals the total number of marbles*

# Alloy Language (1)

---



- An Alloy document is a “source code” unit
- It may contain:
  - ▶ **Signatures**: define types and relationships
  - ▶ **Facts**: properties of models (constraints!)
  - ▶ **Predicates/functions**: reusable expressions
  - ▶ **Assertions**: properties we want to check
  - ▶ **Commands**: instruct the Alloy Analyzer which assertions to check, and how

A predicate is **run** to find a world that satisfies it

An assertion is **checked** to find a counterexample

# Alloy Language (2)

---



- Signatures, predicates, facts and functions tell how correct worlds (models) are made
  - ▶ When the Alloy analyzer tries to build a model, it must comply with them
- Assertions and commands tell which kind of checks must be performed over these worlds
  - ▶ E.g., “find, among all the models, one that violates this assertion”
- Of course, the Analyzer cannot check all the (usually infinite) models of a specification
  - ▶ So you must also tell the Analyzer how to limit the search