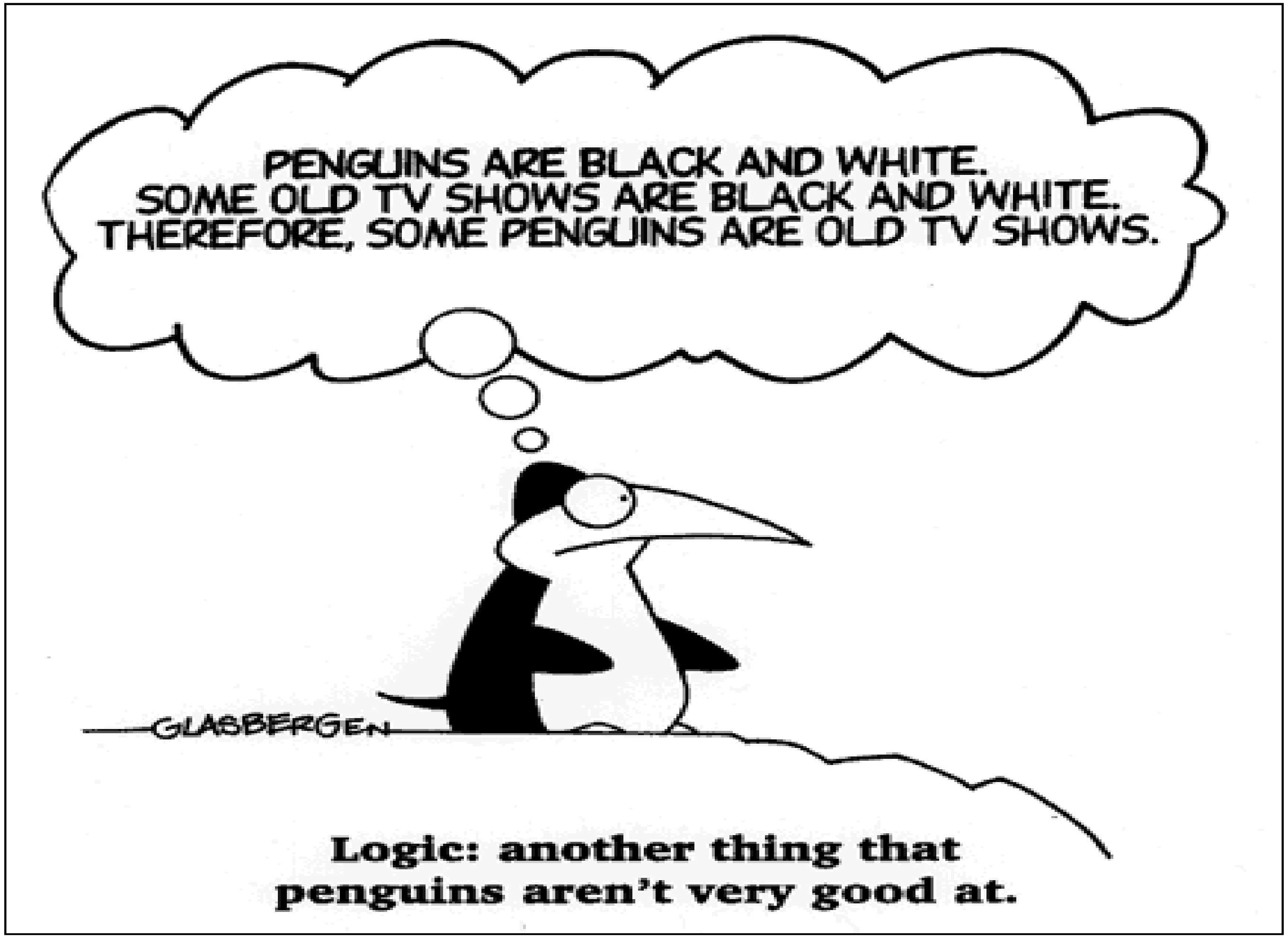
Mathematical Logic



Cartoon copyright Randy Glasbergen (www.glasbergen.com)

Mathematical Logic

- So far we have looked at the language of sets and relations. This is used in Alloy for modelling the *data* representing the state of a software system.
- Now we shall look at the language of logic. This is used to describe logical *properties* of systems.

- Outline:
  - Propositional logic
  - Quantification (predicate logic)
  - Cardinality expressions
  - Let expressions
  - Comprehensions

Propositional Logic Operators

There are two forms of each operator: a shorthand and a verbose form.

Verbose	*Shorthand*	*Operator Name*
not	!	negation
and	&&	conjunction
or	||	disjunction
implies	=>	implication
else	,	alternative
iff	<=>	bi-implication

Propositional Logic Operators

- The first three operators are familiar from programming.
- Their meaning can be defined by the standard truth tables:

p	not p
true	false
false	true

p	q	p and q
true	true	true
true	false	false
false	true	false
false	false	false

p	q	p or q
true	true	true
true	false	true
false	true	true
false	false	false

Implication

- At first glance, the meaning of implication is not so obvious:

  p => q

  p **implies** q

  Common sense interpretation: “if p, then q”

  What sense do these have if p or q are false?

- The truth table for implication:

p	q	p implies q
true	true	true
true	false	false
false	true	true
false	false	true

- Not convinced by the last two rows? Consider these examples:
  – If the moon is made of cheese then 1+1=2.
  – If the moon is made of cheese then I’ll eat my hat!

Else

The **else** operator is used with the implication operator:

F **implies** G **else** H

is equivalent to

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

...G holds when F holds *...H holds when F does not hold*

Bi-implication

The **<=>** operator is two-way implication:

p **iff** q   or   p **<=>** q

is equivalent to

(p **implies** q) **and** (q **implies** p)

This can be read as meaning that p and q are *logically equivalent*. Either they are both true, or they are both false.

Nested implication

Implications can be nested…

C1 => F1,
C2 => F2,
C3 => F3

or equivalently

C1 **implies** F1
else C2 **implies** F2
else C3 **implies** F3

…under condition C1, F1 holds, and if not, then under condition C2, F2 holds, and if not, under condition C3, F3 holds

Syntactic shorthand

Shorthand for conjunction of constraints…

{F G H} …is equivalent to… F **and** G **and** H

The negation symbol can be combined with comparison operators…

a != b …is equivalent to… **not** (a = b)
…is equivalent to… a **not**= b

Equivalent logical expressions

- Each expression on the left is logically equivalent to the expression on the right.

p **and** q	q **and** p
p **and** p	p
not not p	p
not (p **and** q)	(**not** p) **or** (**not** q)
not (p **or** q)	(**not** p) **and** (**not** q)
p **implies** q	(**not** p) **or** q
p **implies** q	(**not** q) **implies** (**not** p)

- For the last one, think about this sentence:

  "*If it's working, then it's hurting*"

Quantified expressions

A quantified expression has the form…

Q x:e | P   where Q is a *quantifier*

The quantifiers used in Alloy are…

all x:e | P	P holds for every x in e;
some x:e | P	P holds for at least one x in e;
no x:e | P	P holds for no x in e;
lone x:e | P	P holds for at most one x in e;
one x:e | P	P holds for exactly one x in e.

Quantified expressions

Several variables can be bound in the same quantifier:

one x: e1, y: e2 | P

... says that there is exactly one combination of values for *x* and *y* that makes P true.

Variables with the same type can share a declaration…

all x, y: e | P

Quantification Examples

```
sig Person
{
  child: set Person
}
```

some p,q: Person| q **in** p.child

...says that there is at least one person who has a child

no p: Person | p **in** p.^child

...says that no person can be reached by following the chain of children from that person (that is there are no cycles in the parent-child relationship)

all p: Person | **some** q: Person | q **in** child.p

...says that every person has at least one parent

Expressing Cardinality

Several quantifier forms can be used to state cardinality constraints on set-valued expressions…

some *s*	*s* contains at least one tuple;
no *s*	*s* contains no tuples (it is empty);
lone *s*	*s* contains at most one tuple;
one *s*	*s* contains exactly one tuple.

Expressing Cardinality

Examples

some Person

…says that the set of persons is not empty;

some child

…says that the child relationship is not empty: there is some pair mapping a person to their child (more succinctly than in the example three slides ago)

no (child.Person - Adult)

…says that only adults can be parents

sig Adult **in** Person { }

all p: Person | **lone** p.child

…says that every person has at most one child (again, more succinctly than in the example three slides ago);

all p: Person | **one** p.child **or** **no** p.child

...says the same thing as the expression above.

Let Expressions

When an expression appears repeatedly, or is a subexpression of a larger, complicated expression, you can factor it out via a let expression.

let x = e | A

means the same as the expression A with each occurrence of x replaced by the expression e.

Let Expressions

Example:

Suppose that, for legal purposes, every person must have at least one heir. If a person has children, they are that person's heirs, otherwise the heir is the queen.

```
sig Person {
  child: set Person,
  heir: some Person
}
one sig queen in Person
```

```
all p: Person |
    let c = p.child |
       (some c  implies p.heir = c else p.heir = queen)
```

This means the same as:

```
all p: Person |
  some p.child
      implies p.heir = p.child
      else p.heir = queen
```

Comprehensions

Comprehensions form relations from properties -- they specify what property a tuple must have for it to belong to a relation.

Syntax: $\{x_1 : e_1, x_2 : e_2, \ldots, x_n : e_n \mid F\}$

Examples:

{p:Adult | **no** p.child.child}

…is the set of adult persons who have no grandchildren

{p,q:Person | q **in** p.^child **or** p **in** q.^child}

…is a relation mapping persons to their family “line” (descendants and ancestors).

Acknowledgements

- Some of the material in this lecture has been adapted (with permission) from slides prepared by John Hatcliff and others for a course on Software Specifications at Kansas State University.