Advanced Object-oriented Programming

Lecture 13

Propositional Logic
Task

Develop a program that:

- allows a user to enter a string representing a term in propositional logic;
- prints out the term with minimal and maximal brackets;
- prints out a truth table for the term.

(actually, we won’t have time for the third one)
Task

Develop a program that:

- allows a user to enter a string representing a term in propositional logic;
- prints out the term with minimal and maximal brackets;
- prints out a truth table for the term.

(actually, we won’t have time for the third one)
Propositional Logic

Propositional logic is the logic of truth and reasoning. It is concerned with how some statements follow logically from other statements to form logical arguments.

It was first systematically studied by Aristotle, and is still of interest in philosophical logic (mainly as one of the simplest non-trivial logics).

It is also relevant to Computer Science as the basis of Boolean logic.
In 1854, Boole published *An Investigation into the Laws of Thought, on Which are founded the Mathematical Theories of Logic and Probabilities*. This work related logical operators and binary arithmetic, and is the basis for what is now called Boolean logic (which is actually just another name for Propositional Logic).
Propositional Logic

Propositional Logic is a *language* consisting of terms that are built from variables, the constants *true* and *false*, and Boolean operators (‘not’, ‘and’, ’or’, etc.).

In BNF:

\[
\langle \text{Prop} \rangle ::= \langle \text{Var} \rangle \mid \text{true} \mid \text{false} \mid \text{not} \langle \text{Prop} \rangle \mid \langle \text{Prop} \rangle \text{and} \langle \text{Prop} \rangle \mid \langle \text{Prop} \rangle \text{or} \langle \text{Prop} \rangle \mid \langle \text{Prop} \rangle \text{implies} \langle \text{Prop} \rangle
\]

\[
\langle \text{Var} \rangle ::= \ 'A \mid 'B \mid 'C \mid \cdots
\]

There is a reason for the single quotes before the variables . . . : we’ll use one of Maude’s built-in types for these.
Propositional Logic

Propositional Logic is a *language* consisting of terms that are built from variables, the constants *true* and *false*, and Boolean operators (‘not’, ‘and’, ’or’, etc.).

In BNF:

\[
\begin{align*}
\langle \text{Prop} \rangle & ::= \langle \text{Var} \rangle \mid \text{true} \mid \text{false} \mid \text{not} \langle \text{Prop} \rangle \mid \\
& \quad \langle \text{Prop} \rangle \text{ and } \langle \text{Prop} \rangle \mid \langle \text{Prop} \rangle \text{ or } \langle \text{Prop} \rangle \mid \\
& \quad \langle \text{Prop} \rangle \text{ implies } \langle \text{Prop} \rangle \\
\langle \text{Var} \rangle & ::= \ 'A \mid 'B \mid 'C \mid \cdots
\end{align*}
\]

There is a reason for the single quotes before the variables . . . : we’ll use one of Maude’s built-in types for these.
Propositional Logic

Propositional Logic is a *language* consisting of terms that are built from variables, the constants *true* and *false*, and Boolean operators (‘not’, ‘and’, ’or’, etc.).

In BNF:

\[
\langle Prop \rangle ::= \langle Var \rangle \mid true \mid false \mid \text{not } \langle Prop \rangle \mid
\langle Prop \rangle \text{ and } \langle Prop \rangle \mid \langle Prop \rangle \text{ or } \langle Prop \rangle \mid
\langle Prop \rangle \text{ implies } \langle Prop \rangle
\]

\[
\langle Var \rangle ::= 'A \mid 'B \mid 'C \mid \cdots
\]

There is a reason for the single quotes before the variables . . . : we’ll use one of Maude’s built-in types for these.
Propositional ADT

Propositions also form an Abstract Data Type.

The propositions themselves are the abstract data values.

The operations are the constants (variables, true, false) and the operators (not, and, or, implies).

We’ll specify this ADT in Maude, then implement it in Java.
Propositional ADT

Propositions also form an Abstract Data Type.
The propositions themselves are the abstract data values.
The operations are the constants (variables, true, false) and the operators (not, and, or, implies).

We’ll specify this ADT in Maude, then implement it in Java.
Propositional ADT

Propositions also form an Abstract Data Type. The propositions themselves are the abstract data values. The operations are the constants (variables, true, false) and the operators (not, and, or, implies).

We’ll specify this ADT in Maude, then implement it in Java.
Propositions also form an Abstract Data Type.

The propositions themselves are the abstract data values.

The operations are the constants (variables, true, false) and the operators (not, and, or, implies).

We’ll specify this ADT in Maude, then implement it in Java.
Maude, Maude, Maude

file: PropLog.maude

```maude
fmod PROP_LOGIC is

*** import Maude's quoted identifiers,
*** renaming the sort Qid to PVar (for "Propositional Variable")
protecting QID *(sort Qid to PVar).
```

Maude's built-in module QID declares a sort Qid of 'quoted identifiers'; these are very like strings, except rather than being enclosed in double quotes ("), they are preceded by a single closing quote ('). E.g., 'a, 'aa, 'abc, etc. This simply means that the sort Qid is now called PVar.
Maude, Maude, Maude

Maude’s built-in module QID declares a sort Qid of ‘quoted identifiers’; these are very like strings, except rather than being *enclosed* in double quotes ("), they are *preceded* by a single closing quote (‘). E.g., ’a, ’aa, ’abc, etc. This simply means that the sort Qid is now called PVar.
Maude, Maude, Maude

fmod PROP_LOGIC is

*** import Maude’s quoted identifiers, 
*** renaming the sort Qid to PVar (for "Propositional Variable")
protecting QID *(sort Qid to PVar) .

Maude’s built-in module QID declares a sort Qid of ‘quoted identifiers’; these are very like strings, except rather than being enclosed in double quotes ("), they are preceded by a single closing quote (’). E.g., ’a, ’aa, ’abc, etc.
This simply means that the sort Qid is now called PVar
Maude, Maude, Maude

```maude
fmod PROP_LOGIC is
  *** import Maude’s quoted identifiers,
  *** renaming the sort Qid to PVar (for "Propositional Variable")
protecting QID *(sort Qid to PVar) .
```

Maude’s built-in module QID declares a sort Qid of ‘quoted identifiers’; these are very like strings, except rather than being *enclosed* in double quotes ("), they are *preceded* by a single closing quote (’). E.g., ’a, ’aa, ’abc, etc. This simply means that the sort Qid is now called PVar
Maude, Maude, Maude

fmod PROP_LOGIC is

*** import Maude’s quoted identifiers,
*** renaming the sort Qid to PVar (for "Propositional Variable")

protecting QID *(sort Qid to PVar) .

Maude’s built-in module QID declares a sort Qid of ‘quoted identifiers’; these are very like strings, except rather than being enclosed in double quotes ("), they are preceded by a single closing quote (’). E.g., ’a, ’aa, ’abc, etc.
This simply means that the sort Qid is now called PVar
Maude, Maude, Maude

file: PropLog.maude

fmod PROP_LOGIC is

*** import Maude's quoted identifiers,
*** renaming the sort Qid to PVar (for "Propositional Variable")
protecting QID *(sort Qid to PVar).

Maude's built-in module QID declares a sort Qid of 'quoted identifiers'; these are very like strings, except rather than being enclosed in double quotes ("), they are preceded by a single closing quote ('). E.g., 'a, 'aa, 'abc, etc. This simply means that the sort Qid is now called PVar.
Maude, Maude, Maude

```maude
fmod PROP_LOGIC is

*** import Maude's quoted identifiers,
*** renaming the sort Qid to PVar (for "Propositional Variable")
protecting QID *(sort Qid to PVar) .
```

Maude's built-in module QID declares a sort Qid of 'quoted identifiers'; these are very like strings, except rather than being *enclosed* in double quotes ("), they are *preceded* by a single closing quote ('). E.g., 'a, 'aa, 'abc, etc.

This simply means that the sort Qid is now called PVar.
Maude, Maude, Maude

```maude
cmd
fmod PROP_LOGIC is

*** import Maude's quoted identifiers,
*** renaming the sort Qid to PVar (for "Propositional Variable")

protecting QID *(sort Qid to PVar).

```

Maude's built-in module QID declares a sort Qid of 'quoted identifiers'; these are very like strings, except rather than being enclosed in double quotes ("), they are preceded by a single closing quote ('). E.g., 'a, 'aa, 'abc, etc.
This simply means that the sort Qid is now called PVar
More Maude

*** propositions

sort Prop.

*** every variable is a proposition

subsort PVar < Prop.

NB this is Maude, not Java; subsorts are not the same thing as inheritance — we’re not suggesting

```
class Prop extends PVar { ... }
```

but it does mean that PVars can be used whenever a Prop is expected.
More Maude

```maude
*** propositions
sort Prop.
*** every variable is a proposition
subsort PVar < Prop.
```

NB this is Maude, not Java; subsorts are *not* the same thing as inheritance — we’re not suggesting

```maude
class Prop extends PVar { ... }
```

but it *does* mean that PVars can be used whenever a Prop is expected.
More Maude

```maude
*** propositions
sort Prop.

*** every variable is a proposition
subsort PVar < Prop.
```

NB this is Maude, not Java; subsorts are not the same thing as inheritance — we’re not suggesting

```java
class Prop extends PVar { ... }
```

but it does mean that PVars can be used whenever a Prop is expected.
**More Maude**

```maude
class Prop extends PVar { ... }
```

NB this is Maude, not Java; subsorts are *not* the same thing as inheritance — we’re not suggesting

but it *does* mean that PVars can be used whenever a Prop is expected.
file PropLog.maude

*** propositions

sort Prop .

*** every variable is a proposition

subsort PVar < Prop .

NB this is Maude, not Java; subsorts are not the same thing as inheritance — we’re not suggesting

class Prop extends PVar { ... }

but it does mean that PVars can be used whenever a Prop is expected.
The first three operations are *infix*; ‘not’ is *prefix*.

Example propositions:

```
not 'a and 'b implies 'c or true
```
**More Maude**

```maude
ops true false : -> Prop [ ctor ] .
op _ and_ : Prop Prop -> Prop [ ctor ] .
op _ or_ : Prop Prop -> Prop [ ctor ] .
op _ implies_ : Prop Prop -> Prop [ ctor ] .
op not_ : Prop -> Prop [ ctor ] .
```

Note: one underscore for each argument

The first three operations are *infix*; ‘not’ is *prefix*

**Example propositions:**

`not 'a and 'b implies 'c or true`
More Maude

```maude
ops true false : -> Prop [ ctor ] .
op _ and_ : Prop Prop -> Prop [ ctor ] .
op _ or_ : Prop Prop -> Prop [ ctor ] .
op _ implies_ : Prop Prop -> Prop [ ctor ] .
op not_ : Prop -> Prop [ ctor ] .
```

Note: one underscore for each argument

The first three operations are infix; ‘not’ is prefix

Example propositions:

not ’a and ’b implies ’c or true
ops true false : -> Prop [ ctor ].

op _ and_ : Prop Prop -> Prop [ ctor ].

op _ or_ : Prop Prop -> Prop [ ctor ].

op _ implies_ : Prop Prop -> Prop [ ctor ].

op not_ : Prop -> Prop [ ctor ].

Note: one underscore for each argument

The first three operations are *infix*; ‘not’ is *prefix*

Example propositions:

\[
\text{not 'a and 'b implies 'c or true}
\]
More Maude

**File:** PropLog.maude

```maude
ops true false : -> Prop [ ctor ] .
op _ and_ : Prop Prop -> Prop [ ctor ] .
op _ or_ : Prop Prop -> Prop [ ctor ] .
op _ implies_ : Prop Prop -> Prop [ ctor ] .
op not_ : Prop -> Prop [ ctor ] .
```

Note: one underscore for each argument

The first three operations are *infix*; ‘not’ is *prefix*

Example propositions:

not ’a and ’b implies ’c or true
More Maude

Note: one underscore for each argument

The first three operations are *infix*; ‘not’ is *prefix*

Example propositions:

not 'a and 'b implies 'c or true
More Maude

file: PropLog.maude

```maude
definitions

ops true false : -> Prop [ ctor ] .
op _ and_ : Prop Prop -> Prop [ ctor ] .
op _ or_ : Prop Prop -> Prop [ ctor ] .
op _ implies_ : Prop Prop -> Prop [ ctor ] .
op not_ : Prop -> Prop [ ctor ] .
```

Note: one underscore for each argument

The first three operations are \textit{infix}; ‘not’ is \textit{prefix}

Example propositions:

```
not 'a and 'b implies 'c or true
```
More Maude

file: PropLog.maude

\begin{itemize}
\item \texttt{ops true false : \rightarrow Prop [ ctor ] .}
\item \texttt{op \_ and\_ : Prop Prop \rightarrow Prop [ ctor ] .}
\item \texttt{op \_ or\_ : Prop Prop \rightarrow Prop [ ctor ] .}
\item \texttt{op \_ implies\_ : Prop Prop \rightarrow Prop [ ctor ] .}
\item \texttt{op not\_ : Prop \rightarrow Prop [ ctor ] .}
\end{itemize}

Note: one underscore for each argument

The first three operations are \textit{infix}; ‘not’ is \textit{prefix}

Example propositions:

\texttt{not 'a and 'b \ implies \ 'c or true}
More Maude

file: PropLog.maude

\begin{verbatim}
ops true false : -> Prop [ ctor ] .
op _ and_ : Prop Prop -> Prop [ ctor ] .
op _ or_ : Prop Prop -> Prop [ ctor ] .
op _ implies_ : Prop Prop -> Prop [ ctor ] .
op not_ : Prop -> Prop [ ctor ] .
\end{verbatim}

Note: one underscore for each argument

The first three operations are \textit{infix}; ‘not’ is \textit{prefix}

Example propositions:

\texttt{not 'a and 'b implies 'c or true}
More Maude

ops true false : -> Prop [ ctor ] .
op _ and_ : Prop Prop -> Prop [ ctor ] .
op _ or_ : Prop Prop -> Prop [ ctor ] .
op _ implies_ : Prop Prop -> Prop [ ctor ] .
op not_ : Prop -> Prop [ ctor ] .

Note: one underscore for each argument

The first three operations are \textit{infix}; ‘not’ is \textit{prefix}

Example propositions:

not 'a and 'b implies 'c or true
More Examples of Terms

- 'a
- true
- not 'a
- not false
- 'a and true
- 'a and not 'b
- 'c or 'a and not 'b
- not 'c implies not 'b and not not 'a

As we can see from the last two examples, terms can be ambiguous:

(‘c or ’a) and not ’b / ‘c or (’a and not ’b)
More Examples of Terms

- 'a
- true
- not 'a
- not false
- 'a and true
- 'a and not 'b
- 'c or 'a and not 'b
- not 'c implies not 'b and not not 'a

As we can see from the last two examples, terms can be ambiguous:

('c or 'a) and not 'b  /  'c or ('a and not 'b)
We can use brackets to disambiguate terms; e.g.,
(‘c or ’a) and not ’b

We can also use conventions so that we sometimes don’t need
brackets to disambiguate terms:
we give a *precedence* to each operator; operations with a low
precedence are more tightly binding.

```maude
op _and_ : Prop Prop -> Prop [ ctor prec 40 ] .
op _or_ : Prop Prop -> Prop [ ctor prec 44 ] .
op _implies_ : Prop Prop -> Prop [ ctor prec 48 ] .
```
We can use brackets to disambiguate terms; e.g.,
(‘c or ’a) and not ’b

We can also use conventions so that we sometimes don’t need brackets to disambiguate terms:
we give a precedence to each operator; operations with a low precedence are more tightly binding.

file: PropLog.maude

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>and</em></td>
<td>Prop Prop -&gt; Prop</td>
<td>ctor prec 40</td>
</tr>
<tr>
<td><em>or</em></td>
<td>Prop Prop -&gt; Prop</td>
<td>ctor prec 44</td>
</tr>
<tr>
<td><em>implies</em></td>
<td>Prop Prop -&gt; Prop</td>
<td>ctor prec 48</td>
</tr>
<tr>
<td>not_</td>
<td>Prop -&gt; Prop</td>
<td>ctor prec 30</td>
</tr>
</tbody>
</table>
We can use brackets to disambiguate terms; e.g.,
(‘c or ’a) and not ’b

We can also use conventions so that we sometimes don’t need brackets to disambiguate terms:
we give a *precedence* to each operator; operations with a low precedence are more tightly binding.

```
file: PropLog.maude

op _and_ : Prop Prop -> Prop [ ctor prec 40 ] .
op _or_  : Prop Prop -> Prop [ ctor prec 44 ] .
op _implies_ : Prop Prop -> Prop [ ctor prec 48 ] .
```
<table>
<thead>
<tr>
<th>Term</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘a and ’b or ’c</td>
<td>(’a and ’b) or ’c</td>
</tr>
<tr>
<td>not ’a and ’b implies ’c</td>
<td>((not ’a) and ’b) implies ’c</td>
</tr>
<tr>
<td>not ’a and (’b implies ’c)</td>
<td>(not ’a) and (’b implies ’c)</td>
</tr>
<tr>
<td>’a and ’b or ’c implies ’d</td>
<td>(’a and ’b) or ’c implies ’d</td>
</tr>
</tbody>
</table>

This disambiguates terms with different operators but what about terms with the same operators, e.g. ‘a and ’b and ’c? 
Examples

<table>
<thead>
<tr>
<th>Term</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>’a and ’b or ’c</td>
<td>(’a and ’b) or ’c</td>
</tr>
<tr>
<td>not ’a and ’b implies ’c</td>
<td>((not ’a) and ’b) implies ’c</td>
</tr>
<tr>
<td>not ’a and (’b implies ’c)</td>
<td>(not ’a) and (’b implies ’c)</td>
</tr>
<tr>
<td>’a and ’b or ’c implies ’d</td>
<td>((’a and ’b) or ’c) implies ’d</td>
</tr>
</tbody>
</table>

This disambiguates terms with different operators

but what about terms with the same operators, e.g.

’a and ’b and ’c  ?
We will adopt the convention that binary operators are right-associative. I.e.,

<table>
<thead>
<tr>
<th>Term</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a or 'b or 'c or 'd or 'e</td>
<td>'a or ('b or ('c or ('d or 'e)))</td>
</tr>
<tr>
<td>'a and 'b and 'c and 'd</td>
<td>'a and ('b and ('c and 'd))</td>
</tr>
<tr>
<td>'a implies 'b implies 'c</td>
<td>'a implies ('b implies 'c)</td>
</tr>
</tbody>
</table>

etc.
Associativity

In fact, we will use a *semantic* property of `and` and `or` (called *associativity*), which says that any way of bracketing

'\text{a}' and '\text{b}' and '\text{c}' and '\text{d}'

gives the same result, so we can omit brackets entirely (and similarly for `or`).

```
file: PropLog.maude

op _and_ : Prop Prop -> Prop [ ctor assoc prec 40 ] .  
op _or_ : Prop Prop -> Prop [ ctor assoc prec 44 ] .  
op _implies_: Prop Prop -> Prop [ ctor prec 48 ] .  
```
Associativity

In fact, we will use a semantic property of \( \text{and} \) and \( \text{or} \) (called associativity), which says that any way of bracketing

\[
'a \text{ and } 'b \text{ and } 'c \text{ and } 'd
\]
gives the same result, so we can omit brackets entirely (and similarly for \( \text{or} \)).

\[
\text{file: PropLog.maude}
\]

\begin{verbatim}
op _and_ : Prop Prop -> Prop [ ctor assoc prec 40 ] .
op _or_ : Prop Prop -> Prop [ ctor assoc prec 44 ] .
op _implies_ : Prop Prop -> Prop [ ctor prec 48 ] .
\end{verbatim}
Printing with Brackets

We still need to specify operations to print with maximal and minimal bracketing.

```maude
op printAllBrackets : Prop -> String .
```

We specify what this operation does by considering what its arguments may be: a `PVar`, `true`, `false`, or a term built from the other four operations (`and`, `or`, `implies`, `not`).
We still need to specify operations to print with maximal and minimal bracketing.

```
file: PropLog.maude

op printAllBrackets : Prop -> String .
```

We specify what this operation does by considering what its arguments may be: a PVar, true, false, or a term built from the other four operations (and, or, implies, not).
Printing with Brackets

file: PropLog.maude

```maude
eq printAllBrackets(true) = "(true)" .
eq printAllBrackets(false) = "(false)" .
var V : PVar .
eq printAllBrackets(V) = "(" + string(V) + ")" .
```

**module QID**

```maude
op string : Qid -> String .
```

**module QID *(sort Qid to PVar)**

```maude
op string : PVar -> String .
```

e.g., string('example) = "example"
Printing with Brackets

file: PropLog.maude

eq printAllBrackets(true) = "(true)" .
eq printAllBrackets(false) = "(false)" .

var V : PVar .
eq printAllBrackets(V) = "(" + string(V) + ")" .

module QID

op string : Qid -> String .

module QID *(sort Qid to PVar)

op string : PVar -> String .

e.g., string('example) = "example"
Printing with Brackets

file: PropLog.maude

```maude
eq printAllBrackets(true) = "(true)" .
eq printAllBrackets(false) = "(false)" .
var V : PVar .
eq printAllBrackets(V) = "(" + string(V) + ")" .
```

module QID

```maude
op string : Qid -> String .
```

module QID *(sort Qid to PVar)

```maude
op string : PVar -> String .
```

e.g., string('example) = "example"
```
Printing with Brackets

eq printAllBrackets(true) = "(true)" .
eq printAllBrackets(false) = "(false)" .

var V : PVar .
eq printAllBrackets(V) = "(" + string(V) + ")" .

module QID

op string : Qid -> String .

module QID *(sort Qid to PVar)

op string : PVar -> String .

e.g., string('example) = "example"
Printing with Brackets

file: PropLog.maude

```maude
eq printAllBrackets(true) = "(true)" .
eq printAllBrackets(false) = "(false)" .
var V : PVar .
eq printAllBrackets(V) = "(" + string(V) + ")" .
```

module QID
```
op string : Qid -> String .
```

module QID *(sort Qid to PVar)
```
op string : PVar -> String .
```

e.g., string('example) = "example"
```
Printing with Brackets

```maude
file: PropLog.maude

eq printAllBrackets(true) = "(true)" .
eq printAllBrackets(false) = "(false)" .

var V : PVar .
eq printAllBrackets(V) = "(" + string(V) + ")" .

module QID

op string : Qid -> String .

module QID *(sort Qid to PVar)

op string : PVar -> String .

e.g., string('example) = "example"
```
Printing with Brackets

```maude
file: PropLog.maude

eq printAllBrackets(true) = "(true)" .
eq printAllBrackets(false) = "(false)" .

var V : PVar .
eq printAllBrackets(V) = "(" + string(V) + ")" .

module QID

op string : Qid -> String .

module QID *(sort Qid to PVar)

op string : PVar -> String .

e.g., string('example) = "example"
```
Printing with Brackets

file: PropLog.maude

\begin{align*}
\text{eq} & \quad \text{printAllBrackets(true)} = "(true)". \\
\text{eq} & \quad \text{printAllBrackets(false)} = "(false)". \\
\text{var} & \quad V : PVar. \\
\text{eq} & \quad \text{printAllBrackets}(V) = "(\text{string}(V)\ +\ )". \\
\end{align*}

module QID

\textbf{op} \quad \text{string} : \text{Qid} \to \text{String}.

\textbf{module QID} *(\text{sort Qid to PVar})

\textbf{op} \quad \text{string} : \text{PVar} \to \text{String}.

e.g., \text{string('example)} = "example"
Printing with Brackets

```
file: PropLog.maude

eq printAllBrackets(true) = "(true)" .

eq printAllBrackets(false) = "(false)" .

var V : PVar .

eq printAllBrackets(V) = "(" + string(V) + ")" .

module QID

op string : Qid -> String .

module QID *(sort Qid to PVar)

op string : PVar -> String .

e.g., string('example) = "example"
```
Printing with Brackets

vars P P1 P2 : Prop .

eq printAllBrackets(P1 and P2) =
    "(" + printAllBrackets(P1) + " and "+ printAllBrackets(P2) + ")" .

*** similarly for ‘or’ and ‘implies’

... 

eq printAllBrackets(not P) = "(not " + printAllBrackets(P) + ")" .

file: PropLog.maude
Printing with Brackets

vars P P1 P2 : Prop .

eq printAllBrackets(P1 and P2) =
  "(" + printAllBrackets(P1) + " and " + printAllBrackets(P2) + ")" .

*** similarly for ‘or’ and ‘implies’

...
Printing with Brackets

vars P P1 P2 : Prop .

eq printAllBrackets(P1 and P2) = 
"(" + printAllBrackets(P1) + " and "
+ printAllBrackets(P2) + ")" .

*** similarly for ‘or’ and ‘implies’

... 

eq printAllBrackets(not P) = "(not " + printAllBrackets(P) + ")" .

file: PropLog.maude
Printing with Brackets

file: PropLog.maude

vars P P1 P2 : Prop.

eq printAllBrackets(P1 and P2) =

"(" + printAllBrackets(P1) + " and " + printAllBrackets(P2) + ")" .

*** similarly for ‘or’ and ‘implies’

...

eq printAllBrackets(not P) = "(not " + printAllBrackets(P) + ")" .
P P1 P2 : Prop.

\[ \text{eq } \text{printAllBrackets}(P1 \text{ and } P2) = \]
\[ "(\text{" + printAllBrackets(P1) + " and " + printAllBrackets(P2) + "\")" .} \]

*** similarly for ‘or’ and ‘implies’

\[ \ldots \]

\[ \text{eq } \text{printAllBrackets}(\text{not } P) = \text{"(not " + printAllBrackets(P) + ")" .} \]
Printing with Brackets

file: PropLog.maude

vars P P1 P2 : Prop.

eq printAllBrackets(P1 and P2) =
"(" + printAllBrackets(P1) + " and " + printAllBrackets(P2) + ")" .

*** similarly for ‘or’ and ‘implies’

...
Printing with Brackets

file: PropLog.maude

vars P P1 P2 : Prop .

eq printAllBrackets(P1 and P2) =
    "(" + printAllBrackets(P1) + " and "
    + printAllBrackets(P2) + ")" .

*** similarly for ‘or’ and ‘implies’

... 

eq printAllBrackets(not P) = "(not " + printAllBrackets(P) + ")" .
Printing with Minimal Brackets

**file: PropLog.maude**

```plaintext
op toString : Prop -> String .
eq toString(true) = "true" .
eq toString(false) = "false" .
eq toString(V) = string(V) .
eq toString(P1 and P2) = ? .
```

If we just write

```plaintext
eq toString(P1 and P2) = toString(P1) + " and " + toString(P2) .
```

(and similarly for or, implies, not)

then we’ll *never* get any brackets anywhere.
Printing with Minimal Brackets

If we just write

```
rule toString(P1 and P2) = toString(P1) + " and " + toString(P2).
```

(and similarly for or, implies, not)

then we’ll never get any brackets anywhere.
Printing with Minimal Brackets

```maude
op toString : Prop -> String .
eq toString(true) = "true" .
eq toString(false) = "false" .
eq toString(V) = string(V) .
eq toString(P1 and P2) = ? .
```

If we just write

```maude
eq toString(P1 and P2) = toString(P1) + " and " + toString(P2) .
```

(and similarly for or, implies, not)

then we’ll *never* get any brackets *anywhere.*
Printing with Minimal Brackets

file: PropLog.maude

\[\text{op } \text{toString} : \text{Prop } \rightarrow \text{String} .\]
\[\text{eq } \text{toString}(\text{true}) = \text{"true"} .\]
\[\text{eq } \text{toString}(\text{false}) = \text{"false"} .\]
\[\text{eq } \text{toString}(\text{V}) = \text{string}(\text{V}) .\]
\[\text{eq } \text{toString}(\text{P1 and P2}) = ? .\]

If we just write
\[\text{eq } \text{toString}(\text{P1 and P2}) = \text{toString}(\text{P1}) + \text{" and "} + \text{toString}(\text{P2}) .\]

(and similarly for or, implies, not)
then we’ll never get any brackets anywhere.
Printing with Minimal Brackets

file: PropLog.maude

```maude
op toString : Prop -> String .
eq toString(true) = "true" .
eq toString(false) = "false" .
eq toString(V) = string(V) .
eq toString(P1 and P2) = ? .
```

If we just write
```
eq toString(P1 and P2) = toString(P1) + " and " + toString(P2) .
```
(and similarly for or, implies, not)
then we’ll never get any brackets anywhere.
Printing with Minimal Brackets

file: PropLog.maude

\begin{itemize}
\item \textbf{op} \hspace{1em} \text{toString} : \text{Prop} \rightarrow \text{String} .
\item \textbf{eq} \hspace{1em} \text{toString}(\text{true}) = "true" .
\item \textbf{eq} \hspace{1em} \text{toString}(\text{false}) = "false" .
\item \textbf{eq} \hspace{1em} \text{toString}(V) = \text{string}(V) .
\item \textbf{eq} \hspace{1em} \text{toString}(P1 \text{ and } P2) = ? .
\end{itemize}

If we just write

\begin{itemize}
\item \textbf{eq} \hspace{1em} \text{toString}(P1 \text{ and } P2) = \text{toString}(P1) + " and " + \text{toString}(P2) .
\end{itemize}

(and similarly for \text{or}, \text{implies}, \text{not})

then we’ll \textit{never} get any brackets \textit{anywhere}. 
Printing with Minimal Brackets

file: PropLog.maude

\begin{verbatim}
op toString : Prop -> String .

eq toString(true) = "true" .

eq toString(false) = "false" .

eq toString(V) = string(V) .

eq toString(P1 and P2) = ? .
\end{verbatim}

If we just write

\begin{verbatim}
eq toString(P1 and P2) = toString(P1) + " and " + toString(P2) .
\end{verbatim}

(and similarly for or, implies, not)

then we’ll never get any brackets anywhere.
Printing with Minimal Brackets

```maude
eq 
toString(P1 and P2) = ? .
```

If \( P1 \) is of the form \( P3 \) and \( P4 \), then we don’t need brackets around the first argument.

E.g.,

```
toString('a and 'b and 'c) = "'a and 'b and 'c" .
```

But if \( P1 \) is of the form \( P3 \) or \( P4 \), then we do need brackets around the first argument.

E.g.,

```
toString('a or 'b and 'c) = "('a or 'b) and 'c" .
```
Printing with Minimal Brackets

file: PropLog.maude

eq toString(P1 and P2) = ? .

If $P_1$ is of the form $P_3$ and $P_4$, then we don’t need brackets around the first argument.

E.g.,

$\text{toString('a and 'b and 'c)} = "'a and 'b and 'c" .$

But if $P_1$ is of the form $P_3$ or $P_4$, then we do need brackets around the first argument.

E.g.,

$\text{toString('a or 'b and 'c)} = "('a or 'b) and 'c" .)$
**Printing with Minimal Brackets**

If \( P_1 \) is of the form \( P_3 \) and \( P_4 \), then we don’t need brackets around the first argument.

E.g.,

\[
\text{toString('a and 'b and 'c)} = "'a and 'b and 'c"
\]

But if \( P_1 \) is of the form \( P_3 \) or \( P_4 \), then we do need brackets around the first argument.

E.g.,

\[
\text{toString('a or 'b and 'c)} = "('a or 'b) and 'c"
\]
In the second case, brackets were needed because `or` has lower precedence than `and`.

We’ll add an extra parameter that represents ‘the precedence of the operator above a term’.

This gives us exactly the information we need in order to decide whether an operand requires brackets around it.

```maude
*** put brackets around the proposition if the precedence of
*** its main operator is greater than the given integer

op toStringPrec : Prop Int -> String .
```
In the second case, brackets were needed because or has lower precedence than and.

We’ll add an extra parameter that represents ‘the precedence of the operator above a term’.

This gives us exactly the information we need in order to decide whether an operand requires brackets around it.

file: PropLog.maude

```plaintext
*** put brackets around the proposition if the precedence of
*** its main operator is greater than the given integer

op toStringPrec : Prop Int -> String .
```
Printing with Minimal Brackets

In the second case, brackets were needed because or has lower precedence than and.

We’ll add an extra parameter that represents ‘the precedence of the operator above a term’.

This gives us exactly the information we need in order to decide whether an operand requires brackets around it.

```maude
file: PropLog.maude

*** put brackets around the proposition if the precedence of
*** its main operator is greater than the given integer

op toStringPrec : Prop Int -> String .
```
Printing with Minimal Brackets

In the second case, brackets were needed because `or` has lower precedence than `and`.

We’ll add an extra parameter that represents ‘the precedence of the operator above a term’.

This gives us exactly the information we need in order to decide whether an operand requires brackets around it.

```maude
file: PropLog.maude

*** put brackets around the proposition if the precedence of
*** its main operator is greater than the given integer

op toStringPrec : Prop Int -> String .
```
Printing with Minimal Brackets

file: PropLog.maude

*** no operator has precedence greater than 48,
*** so no outermost brackets

eq toString(P) = toStringPrec(P, 48).

var I : Int.

eq toStringPrec(true, I) = "true".
eq toStringPrec(false, I) = "false".
eq toStringPrec(V, I) = string(V).
Printing with Minimal Brackets

file: PropLog.maude

*** no operator has precedence greater than 48,
*** so no outermost brackets

eq toString(P) = toStringPrec(P, 48) .

var I : Int .

eq toStringPrec(true, I) = "true" .

eq toStringPrec(false, I) = "false" .

eq toStringPrec(V, I) = string(V) .
Printing with Minimal Brackets

*** no operator has precedence greater than 48, 
*** so no outermost brackets

eq \text{toString}(P) = \text{toStringPrec}(P, 48).

\text{var} I : \text{Int}.

eq \text{toStringPrec}(\text{true}, I) = "true".

eq \text{toStringPrec}(\text{false}, I) = "false".

\text{eq} \text{toStringPrec}(V, I) = \text{string}(V).
Printing with Minimal Brackets

*** no operator has precedence greater than 48, 
*** so no outermost brackets

\[
\text{eq } \text{toString}(P) = \text{toStringPrec}(P, 48) .
\]

\[
\text{var } I : \text{Int} .
\]

\[
\text{eq } \text{toStringPrec}(\text{true}, I) = "true" .
\]

\[
\text{eq } \text{toStringPrec}(\text{false}, I) = "false" .
\]

\[
\text{eq } \text{toStringPrec}(V, I) = \text{string}(V) .
\]
Printing with Minimal Brackets

file: PropLog.maude

cq
toStringPrec(P1 and P2, I) =
    toStringPrec(P1, 40) + " and " + toStringPrec(P2, 40)
if I >= 40 .  *** no brackets needed

cq
toStringPrec(P1 and P2, I) =
    "(" + toStringPrec(P1, 40) + " and " + toStringPrec(P2, 40) + ")"
if I < 40 .  *** brackets are needed

so P1 and P2 will have brackets if needed
similarly for or and not
Printing with Minimal Brackets

```maude
file: PropLog.maude

cq toStringPrec(P1 and P2, l) =
    toStringPrec(P1, 40) + " and " + toStringPrec(P2, 40)
if l >= 40 . *** no brackets needed

cq toStringPrec(P1 and P2, l) =
    "(" + toStringPrec(P1, 40) + " and " + toStringPrec(P2, 40) + ")"
if l < 40 . *** brackets are needed

so P1 and P2 will have brackets if needed
similarly for or and not
```
Printing with Minimal Brackets

```maude
cq  toStringPrec(P1 and P2, I) =
    toStringPrec(P1, 40) + " and " + toStringPrec(P2, 40)
if I >= 40 . *** no brackets needed

cq  toStringPrec(P1 and P2, I) =
    "(" + toStringPrec(P1, 40) + " and " + toStringPrec(P2, 40) + ")"
if I < 40 . *** brackets are needed
```

so P1 and P2 will have brackets if needed
similarly for or and not
Printing with Minimal Brackets

**file:** PropLog.maude

cq $toStringPrec(P1 \text{ and } P2, I) =$
$toStringPrec(P1, 40) + " and " + toStringPrec(P2, 40)$
if $I \geq 40$ . *** no brackets needed

cq $toStringPrec(P1 \text{ and } P2, I) =$
"(" + $toStringPrec(P1, 40) + " and " + toStringPrec(P2, 40) + ")"
if $I < 40$ . *** brackets are needed

so $P1$ and $P2$ will have brackets if needed
similarly for or and not
Printing with Minimal Brackets

cq \text{toStringPrec}(P1 \text{ and } P2, I) =
\text{toStringPrec}(P1, 40) + " and " + \text{toStringPrec}(P2, 40)
\text{if } I \geq 40. \text{ *** no brackets needed}

cq \text{toStringPrec}(P1 \text{ and } P2, I) =
"(" + \text{toStringPrec}(P1, 40) + " and " + \text{toStringPrec}(P2, 40)
+ ")"
\text{if } I < 40. \text{ *** brackets are needed}

so \text{P1 and P2} will have brackets if needed
similarly for \text{or and not}
Printing with Minimal Brackets

file: PropLog.maude

\texttt{cq}
\begin{align*}
\text{toStringPrec}(P1 ~\text{and}~ P2, I) &= \\
&\text{toStringPrec}(P1, 40) + " ~\text{and}~ " + \text{toStringPrec}(P2, 40) \\
\text{if} & \ I \geq 40 \ . \ \text{*** no brackets needed}
\end{align*}

\texttt{cq}
\begin{align*}
\text{toStringPrec}(P1 ~\text{and}~ P2, I) &= \\
&"(~ + \text{toStringPrec}(P1, 40) + ~\text{and}~ " + \text{toStringPrec}(P2, 40) \\
&+ ~")"
\text{if} & \ I < 40 \ . \ \text{*** brackets are needed}
\end{align*}

so \( P1 \) and \( P2 \) will have brackets if needed
similarly for \texttt{or} and \texttt{and} \texttt{not}
Printing with Minimal Brackets

```maude
cq  toStringPrec(P1 and P2, I) =
    toStringPrec(P1, 40) + " and " + toStringPrec(P2, 40)
if I >= 40 . *** no brackets needed

cq  toStringPrec(P1 and P2, I) =
    "(" + toStringPrec(P1, 40) + " and " + toStringPrec(P2, 40) + ")"
if I < 40 . *** brackets are needed
```

so P1 and P2 will have brackets if needed
similarly for or and not
Printing with Minimal Brackets

---

file: PropLog.maude

cq toStringPrec(P1 and P2, I) =
    toStringPrec(P1, 40) + " and " + toStringPrec(P2, 40)
if I >= 40 . *** no brackets needed

cq toStringPrec(P1 and P2, I) =
    "(" + toStringPrec(P1, 40) + " and " + toStringPrec(P2, 40) + ")"
if I < 40 . *** brackets are needed

so P1 and P2 will have brackets if needed
similarly for or and not
Printing with Minimal Brackets

file: PropLog.maude

cq  tostringPrec(P1 and P2, I) =
    tostringPrec(P1, 40) + " and " + tostringPrec(P2, 40)
if  I >= 40 . *** no brackets needed

cq  tostringPrec(P1 and P2, I) =
    "(" + tostringPrec(P1, 40) + " and " + tostringPrec(P2, 40)
    + ")"
if  I < 40 . *** brackets are needed

so P1 and P2 will have brackets if needed
similarly for or and not
Printing with Minimal Brackets

cq  toStringPrec(P1 implies P2, I) =
    toStringPrec(P1, 47) + " implies " + toStringPrec(P2, 48)
if I >= 48 . *** no brackets needed

cq  toStringPrec(P1 implies P2, I) =
    "(" + toStringPrec(P1, 47) + " implies " + toStringPrec(P2, 48) + "")"
if I < 48 . *** brackets are needed

This gives us right associativity:
P1 will have brackets round it if its main operator is ‘implies’;
P2 will not
Printing with Minimal Brackets

```plaintext
file: PropLog.maude

cq  toStringPrec(P1 implies P2, I) =
    toStringPrec(P1, 47) + " implies " + toStringPrec(P2, 48)
if  I >= 48 .  *** no brackets needed

cq  toStringPrec(P1 implies P2, I) =
    "(" + toStringPrec(P1, 47) + " implies " + toStringPrec(P2, 48) + ")"
if  I < 48 .  *** brackets are needed
```

This gives us right associativity:
P1 will have brackets round it if its main operator is ‘implies’;
P2 will not
Printing with Minimal Brackets

cq  toStringPrec(P1 implies P2, I) =
    toStringPrec(P1, 47) + " implies " + toStringPrec(P2, 48)
if  I >= 48 .  *** no brackets needed

cq  toStringPrec(P1 implies P2, I) =
    "(" + toStringPrec(P1, 47) + " implies " + toStringPrec(P2, 48) + ")"
if  I < 48 .  *** brackets are needed

This gives us right associativity:
P1 will have brackets round it if its main operator is ‘implies’;
P2 will not
Printing with Minimal Brackets

 cq  tostringPrec(P1 implies P2, I) =
   tostringPrec(P1, 47) + " implies " + tostringPrec(P2, 48)
 if  I >= 48 .  *** no brackets needed

cq  tostringPrec(P1 implies P2, I) =
   "(" + tostringPrec(P1, 47) + " implies " + tostringPrec(P2, 48) + ")"
 if  I < 48 .  *** brackets are needed

This gives us right associativity:
P1 will have brackets round it if its main operator is ‘implies’; P2 will not
Printing with Minimal Brackets

This gives us right associativity:
P1 will have brackets round it if its main operator is ‘implies’;
P2 will not
Printing with Minimal Brackets

file: PropLog.maude

cq \text{toStringPrec}(P1 \text{ implies } P2, I) =
\text{toStringPrec}(P1, 47) + " \text{ implies } " + \text{toStringPrec}(P2, 48)
if I \geq 48 . *** no brackets needed

cq \text{toStringPrec}(P1 \text{ implies } P2, I) =
"(" + \text{toStringPrec}(P1, 47) + " \text{ implies } " + \text{toStringPrec}(P2, 48) + ")"
if I < 48 . *** brackets are needed

This gives us right associativity:
P1 will have brackets round it if its main operator is ‘implies’;
P2 will not
Printing with Minimal Brackets

\[
\text{toStringPrec}(P1 \implies P2, I) =
\begin{cases}
\text{toStringPrec}(P1, I) + " \implies " + \text{toStringPrec}(P2, I) & \text{if } I \geq 48 . \quad *** \text{no brackets needed} \\
"(" + \text{toStringPrec}(P1, I) + " \implies " + \text{toStringPrec}(P2, I) + ")" & \text{if } I < 48 . \quad *** \text{brackets are needed}
\end{cases}
\]

This gives us right associativity: P1 will have brackets round it if its main operator is ‘implies’; P2 will not.
Printing with Minimal Brackets

cq  toStringPrec(P1 implies P2, I) =
    toStringPrec(P1, 47) + " implies " + toStringPrec(P2, 48)
if  I >= 48 .  *** no brackets needed

cq  toStringPrec(P1 implies P2, I) =
    "(" + toStringPrec(P1, 47) + " implies " + toStringPrec(P2, 48) + ")"
if  I < 48 .  *** brackets are needed

This gives us right associativity:
P1 will have brackets round it if its main operator is ‘implies’;
P2 will not
reduce tostring(('a or 'b) and 'c).

= 
  tostringPrec(('a or 'b) and 'c, 48)
=  *** 48 >= 40; no brackets needed
tostringPrec('a or 'b, 40) + " and " + tostringPrec('c, 40)
=  *** 40 < 44; brackets are needed
  "(" + tostringPrec('a, 44) + " or " + tostringPrec('b, 44) + ")" + " and " + tostringPrec('c, 40)
=  "(" + string('a) + " or " + string('b) + ")" + " and " + string('c)
=  "(a or b) and c"
reduce tostring(('a or 'b) and 'c) .

= 

toStringPrec(('a or 'b) and 'c, 48)

=  *** 48 >= 40; no brackets needed  
toStringPrec('a or 'b, 40) + " and " + toStringPrec('c, 40)

=  *** 40 < 44; brackets are needed  
"(" + toStringPrec('a, 44) + " or " + toStringPrec('b, 44) + ")"
+ " and " + toStringPrec('c, 40)

=  

"(" + string('a) + " or " + string('b) + ")"
+ " and " + string('c)

=  

"(a or b) and c"
reduce toString(('a or 'b) and 'c).

= toStringPrec(('a or 'b) and 'c, 48)

= *** 48 >= 40; no brackets needed
  toStringPrec('a or 'b, 40) + " and " + toStringPrec('c, 40)

= *** 40 < 44; brackets are needed
  "(" + toStringPrec('a, 44) + " or " + toStringPrec('b, 44) + ")"
  + " and " + toStringPrec('c, 40)

= "(" + string('a) + " or " + string('b) + ")"
  + " and " + string('c)

= "(a or b) and c"
reduce  toString(('a or 'b) and 'c) .

=  toStringPrec(('a or 'b) and 'c, 48)

=  *** 48 >= 40; no brackets needed
   toStringPrec('a or 'b, 40) + " and " + toStringPrec('c, 40)

=  *** 40 < 44; brackets are needed
   "(" + toStringPrec('a, 44) + " or " + toStringPrec('b, 44) + ")"
   + " and " + toStringPrec('c, 40)

=  "(" + string('a) + " or " + string('b) + ")"
   + " and " + string('c)

=  "(a or b) and c"
reduce toString(('a or 'b) and 'c) .

= toStringPrec(('a or 'b) and 'c, 48)

= *** 48 >= 40; no brackets needed
   toStringPrec('a or 'b, 40) + " and " + toStringPrec('c, 40)

= *** 40 < 44; brackets are needed
   "(" + toStringPrec('a, 44) + " or " + toStringPrec('b, 44) + ")"
   + " and " + toStringPrec('c, 40)

= "(" + string('a) + " or " + string('b) + ")"
   + " and " + string('c)

= "(a or b) and c"
reduce  toString(('a or 'b) and 'c) .

=  
  toStringPrec(('a or 'b) and 'c, 48)
=  *** 48 >= 40; no brackets needed
  toStringPrec('a or 'b, 40) + " and " + toStringPrec('c, 40)
=  *** 40 < 44; brackets are needed
  "(" + toStringPrec('a, 44) + " or " + toStringPrec('b, 44) + ")"
  + " and " + toStringPrec('c, 40)
=  
  "(" + string('a) + " or " + string('b) + ")"
  + " and " + string('c)
=  
  "(a or b) and c"
**reduce**  \( \text{toString}((\text{'}a \text{ or } \text{'b}) \text{ and } \text{'c}) \). \\

\[
= \\
\text{toStringPrec}((\text{'}a \text{ or } \text{'b}) \text{ and } \text{'c}, 48) \\
= \textbf{*** 48 >= 40; no brackets needed} \\
\text{toStringPrec('a or 'b, 40) + ' and ' + toStringPrec('c, 40)} \\
= \textbf{*** 40 < 44; brackets are needed} \\
'(" + toStringPrec('a, 44) + ' or ' + toStringPrec('b, 44) + ")" \\
+ ' and ' + toStringPrec('c, 40) \\
= \\
'(" + string('a) + ' or ' + string('b) + ")" \\
+ ' and ' + string('c) \\
= \\
'(a or b) and c"
reduce  toString(('a or 'b) and 'c) .

=  toStringPrec(('a or 'b) and 'c, 48)

=  *** 48 >= 40; no brackets needed
   toStringPrec('a or 'b, 40) + " and " + toStringPrec('c, 40)

=  *** 40 < 44; brackets are needed
   "(" + toStringPrec('a, 44) + " or " + toStringPrec('b, 44) + ")"
   + " and " + toStringPrec('c, 40)

=  "(" + string('a) + " or " + string('b) + ")"
   + " and " + string('c)

=  "(a or b) and c"
reduce  toString(('a or 'b) and 'c) .

=  
toStringPrec(('a or 'b) and 'c, 48)

=  *** 48 >= 40; no brackets needed
   toStringPrec('a or 'b, 40) + " and " + toStringPrec('c, 40)

=  *** 40 < 44; brackets are needed
   "(" + toStringPrec('a, 44) + " or " + toStringPrec('b, 44) + ")"
   + " and " + toStringPrec('c, 40)

=  
   "(" + string('a) + " or " + string('b) + ")"
   + " and " + string('c)

=  
   "(a or b) and c"
reduce toString(’a or ’b) and ’c) .

= 

toStringPrec(’a or ’b) and ’c, 48)

= *** 48 >= 40; no brackets needed

toStringPrec(’a or ’b, 40) + " and " + toStringPrec(’c, 40)

= *** 40 < 44; brackets are needed

"(" + toStringPrec(’a, 44) + " or " + toStringPrec(’b, 44) + ")" + " and " + toStringPrec(’c, 40)

= 

"(" + string(’a) + " or " + string(’b) + ")" + " and " + string(’c)

= 

"(a or b) and c"
reduce  toString(('a or 'b) and 'c) .

=  
   toStringPrec(('a or 'b) and 'c, 48)
=  *** 48 >= 40; no brackets needed
   toStringPrec('a or 'b, 40) + " and " + toStringPrec('c, 40)
=  *** 40 < 44; brackets are needed
   "(" + toStringPrec('a, 44) + " or " + toStringPrec('b, 44) + ")"
   + " and " + toStringPrec('c, 40)
=  
   "(" + string('a) + " or " + string('b) + ")"
   + " and " + string('c)
=  
   "(a or b) and c"
public class Prop {
    public Prop and(Prop p1, Prop p2) { ... }
    // similarly for `or`, etc.
    public String printAllBrackets() { ... }
    public String toString() { ... }
}

We need a data representation.
We need a data representation.
public class Prop {
    public Prop and(Prop p1, Prop p2) { ... }
    // similarly for 'or', etc.
    public String printAllBrackets() { ... }
    public String toString() { ... }
}

We need a data representation.
That’s All, Folks!

Summary

- sort Prop
- Maude equations
- helper functions

Next:

data representation