Project Report - Beta

N. Cohen, T. Coroneo, V. Iyer, G. Kaminer, and L. Madison

Friday 28th March, 2025

Abstract

We implement an explicit model checker for bilateral state-based modal logic (BSML), as described in [?]. BSML has been used to account for Free-Choice (FC) and related inferences which arise from speakers' distaste for interpretations that verify a sentence by empty configuration (*neglect-zero tendency*).

Free-Choice (FC) inferences are instances of conjunctive meaning being unexpectedly derived from a disjunctive sentence. In the following example, a modalized disjunction yields a conjunction of modals.

- You may go to the beach or to the cinema $\rightsquigarrow$ You may go to the beach and you may go to the cinema

- $\Diamond(b \vee c) \rightsquigarrow \Diamond b \wedge \Diamond c$

Aloni (2022) explain that in interpreting a sentence, a speaker identifies which are the structures of reality that would reflect the sentence (the models in which the sentence would be assertable). In our disjunction, there are four associated worlds:

1. W_{bc} in which both b and c are true (you go to both the beach and the cinema)

2. W_b in which b is true (you go to the beach)

3. W_c in which c is true (you go to the cinema)

4. W_z in which neither b nor c is true (you don't go anywhere)

We can use BSML to model the information states (i.e. sets of worlds) in which the disjunction $b \vee c$ would be assertable. A proposition is assertable at a state s if it is assertable in all its substates. A disjunction is assertable in a state s if each of the disjuncts is supported in a substate of s.

1. In a state consisting of W_b and W_c, $b \vee c$ is assertable

2. In a state consisting of W_{bc} and W_b (or a state consisting of W_{bc} and W_c), $b \vee c$ is assertable

3. In a state consisting of W_b (or a state consisting of W_c), $b \vee c$ is assertable since each of the disjuncts is supported in a substate. b is supported in W_b, and c is supported in the empty state.

4. In a state consisting of W_z and W_b (and any other state which includes W_z), $b \vee c$ is *not* assertable because in W_z both b and c are false, so this state would leave open the possibility that neither b nor c is the case.

So among the four types of states, $b \vee c$ is assertable in all but the last. This is a problem, because if $b \vee c$ is assertable In a state consisting of only W_b (or a state consisting of only W_c) then we have that $\diamond(b \vee c)$ is true while $\diamond b \wedge \diamond c$ is false, so the FC inference fails. The problematic state, then, is the zero-model: one of the states which it uses to satisfy the disjunction is the empty state.

How do we account for FC inferences then? Aloni argues pragmatically that a speaker would not consider the zero-model as one of the candidate states. Neglecting the zero model then, the FC inference would hold because the only states that would support $b \vee c$ would be (1) or (2). To model neglect-zero (to make sure that $b \vee c$ is not assertable in the zero-model), we require that to satisfy a disjunction, the state must be the union of two non-empty substates rather than just the union of two substates. So we enrich each formula with a $[]^+$ function which conjuncts the non-emptiness atom (NE) to each formula and all its subformulas.

After enrichment, the disjunction $b \vee c$ is no longer assertable in a state consisting of only W_b (or a state consisting of only W_c) since the non-emptiness atom (NE) would not be supported in all the substates. Finally then, if the only states in which the disjunction holds are (1) and (2), then the FC inference holds.

Contents

1 How to use this?

To generate the PDF, open `report.tex` in your favorite LATEXeditor and compile. Alternatively, you can manually do `pdflatex report; bibtex report; pdflatex report; pdflatex report` in a terminal.

You should have stack installed (see `https://haskellstack.org/`) and open a terminal in the same folder.

- To compile everything: `stack build`.

- To open ghci and play with your code: `stack ghci`

- To run the executable from Section **??**: `stack build && stack exec myprogram`

- To run the tests from Section **??**: `stack clean && stack test --coverage`

1.1 Syntax

```haskell
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE RankNTypes #-}
module Syntax where

import Data.Data (Data)

import Control.Lens
import Control.Lens.Extras (is)

import Test.QuickCheck
```

This module describes the syntactical elements of BSML. We define formulas, some syntactical operations and provide generators for random formulas.

The formulas of BSML are defined as follows:

```haskell
type Proposition = Int

-- Formulas of BSML.
data Form
  = Bot
  | NE
  | Prop Proposition
  | Neg Form
  | And Form Form
  | Or  Form Form
  | Gor Form Form
  | Dia Form
  deriving (Eq,Ord,Show,Data)
```

Readers familiar with Modal Logic (see e.g.[**?**]) should recognize this as the basic modal language, extended with `NE`, the nonemptiness atom, and , global disjunction. As we will see when defining the semantics, `NE`is used to exclude the assertion of logical statements due to empty information-configurations (i.e. empty states/teams).

For the sake of more legible output, we also define a pretty-printer for formulas. Note that we print as `V/` rather than `\V` to avoid having to worry about the escape character.

```haskell
ppForm :: Form -> String
ppForm = \case
  Bot       -> "_|_"
  NE        -> "NE"
  Prop p    -> show p
  Neg f     -> "~" ++ ppForm f
  And f1 f2 -> "(" ++ ppForm f1 ++ " & " ++ ppForm f2 ++ ")"
  Or f1 f2  -> "(" ++ ppForm f1 ++ " v " ++ ppForm f2 ++ ")"
  Gor f1 f2 -> "(" ++ ppForm f1 ++ " V/ " ++ ppForm f2 ++ ")"
  Dia f     -> "<>" ++ ppForm f
```

Further, we define some abbreviations for formulas, following [?]. As usual, $\Box$ is defined as the dual of $\Diamond$. As will become evident when we define the semantics, $\bot$ is supported in a state if and only if that state is empty. It is therefore referred to as the *weak contradiction*. The strong contradiction, defined as $\bot\!\!\!\bot := \bot \wedge \texttt{NE}$, will never be supported. Dually, the formula $\top := \texttt{NE}$ serves as the *weak tautology*, being supported by non-empty states and the **strong tautology** $\top\!\!\!\top := \neg\bot$ is always supported.

```haskell
-- Define box as the dual of diamond.
box :: Form -> Form
box = Neg . Dia . Neg

-- Define the strong contradiction (which is never assertable).
botbot :: Form
botbot = And Bot NE

-- NE functions as a weak tautology (assertable in non-empty states).
top :: Form
top = NE

-- Define the strong tautology (which is always assertable).
toptop :: Form
toptop = Neg Bot
```

As in [?], we can use these notions of contradiction and tautology to interpret finite (global) disjunctions and conjunctions:

```haskell
bigOr :: [Form] -> Form
bigOr [] = Bot
bigOr fs = foldr1 Or fs

bigAnd :: [Form] -> Form
bigAnd [] = toptop
bigAnd fs = foldr1 And fs

bigGor :: [Form] -> Form
bigGor [] = botbot
bigGor fs = foldr1 Gor fs
```

Note that we could have (semantically equivalently) defined e.g.

```haskell
bigor :: [Form] -> Form
bigor = foldr Or Bot
```

but this would have had the undesired side-effect of including Bot in *every* disjunction, including non-empty ones.

1.1.1 Random formulas

In order to verify some properties of BSML, we would like to be able to generate random formulas. We will use QuickCheck's ecosystem for this purpose, so we only need to define an instance of `Arbitrary` for `Form`.

First, we fix a number of propositions, which we will also use when generating random valuations for models. This guarantees that random models and formulas have a more meaningful interaction, in the sense that the formulas will actually refer to the propositions that occur in the model.

```
-- We use proposition in the range (1, numProps).
numProps :: Int
numProps = 32
```

One might wonder why we do not use the size parameter of a generator to determine the range of propositions. We intentionally avoid this, because it would introduce a bias in the occurence of Propositions, where more nested subformulas will not contain high propositions.

Now we can define the `Arbitrary Form`-instance using a standard sized generator, where formulas generated with size 0 are random atoms and larger formulas are generated by applying a random constructor to random smaller formulas.

```
-- Generate a random atom.
randomAtom :: Gen Form
randomAtom = oneof [Prop <$> choose (1, numProps), pure NE, pure Bot]

instance Arbitrary Form where
  arbitrary = sized $ \case
    0 -> randomAtom
    _ ->  oneof [
        randomAtom,
        Neg <$> f,
        And <$> f <*> f,
        Or  <$> f <*> f,
        Gor <$> f <*> f,
        Dia <$> f
      ]
    where f = scale (`div` 2) arbitrary
```

The choice to scale the size of the generator by dividing it by 2 is completely arbitrary, but seems to work well in practice and is used in similar projects, see e.g. [?].

Last, we also define shrinks of formulas that empower QuickCheck to attempt simplifying counterexamples when/if it finds any.

```
  shrink (Neg f)    = [Bot, NE, f]     ++ [Neg f'     | f'      <- shrink f]
  shrink (Dia f)    = [Bot, NE, f]     ++ [Dia f'     | f'      <- shrink f]
  shrink (And f1 f2) = [Bot, NE, f1, f2] ++ [And f1' f2' | (f1',f2') <- shrink (f1,f2)]
  shrink (Or f1 f2)  = [Bot, NE, f1, f2] ++ [Or  f1' f2' | (f1',f2') <- shrink (f1,f2)]
  shrink (Gor f1 f2) = [Bot, NE, f1, f2] ++ [Gor f1' f2' | (f1',f2') <- shrink (f1,f2)]
  shrink _          = []
```

1.1.2   Boilerplate for subformulas

This section is slightly technical and can safely be skipped. It introduces some functions that allow us to check properties of subformulas (e.g. whether a formula contains `NE` anywhere in its subformulas).

The `Lens`-library defines a typeclass `Plated` that implements a lot of boilerplate code for the transitive descendants of values of recursively defined types, so let us make `Form` an instance.

```
-- Use default implementation: plate = uniplate
instance Plated Form
```

We also derive a prism corresponding to every constructor of `Form`.

```
-- Derives prisms _Bot, _NE, ..., _Gor, _Dia
makePrisms ''Form
```

For readers unfamiliar with this, a`Prism` is to a constructors what a `Lens` is to a field. For example, the derived `Prism` for `Or` is of type

```
_Or :: Prism' Form (Form, Form)
```

which can loosely be interpreted as a pair of functions; one that turns a `(Form, Form)` into a `Form` (by applying `Or` in this case), and one that tries to turn a `Form` into a `(Form, Form)` (in this case, by taking the arguments out of the constructor if the formula is a disjunction). As we are familiar with from `Lens`, this is suitably generalized.

Now, we can e.g. check whether a formula uses the constructors `NE` or `Gor` by seeing if any of its transitive descendants is built using one of these constructors.

```
isBasic :: Form -> Bool
isBasic = any ((||) . is _NE <*> is _Gor) . universe
```

Or more generally, check whether a certain constructor was used.

```
hasCr :: Prism' Form fs -> Form -> Bool
hasCr = (. universe) . any . is
```

1.2   Semantics

We quickly recall the most important aspects of the semantics of BSML from [**?**]. We interpret formulas on (Kripke) models, which consist of

- a set of worlds W;

- a binary relation $R \subseteq W \times W$ between worlds;

- and a valuation $V : W \to \wp(\texttt{Prop})$ mapping a world to the propositions that hold in it. [1]

A *team* or *state* (we will use these terms interchangably) on a model is a subset $s \subseteq W$. To link back to the introduction, the worlds represent information-configurations and a team represents the worlds that a speaker perceives as possible.

As the name *Bilateral State-based Modal Logic* suggests, formulas are evaluated with respect to a team (rather than a world). The "bilateral" part refers to the fact that we make use of *two* fundamental semantics notions; *support* and *anti-support*, rather than just *truth*. Support (resp. antisupport) of a formula by a team represents the speaker's ability to assert (resp. reject) the formula, given the worlds deemed possible in the team.

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}
module Semantics where

import Syntax

import Data.List
import Data.Set (Set)
import Data.IntMap (IntMap)
import qualified Data.Set as Set
import qualified Data.IntMap as IntMap

import Test.QuickCheck
```

---

[1]In the paper, the valuation is defined as a function $\texttt{Prop} -> \wp W$, but this is easily seen to be equivalent.

In our implementation, we will represent worlds as `Int`'s and teams as lists of worlds. For the relation and valuation, we use `IntMap`'s, which are conceptually identical to association lists with `Int`-valued keys, but allow for fast lookup of the successors and satisfied proposition letters of a world.

```
type World = Int
type Team = [World]
type Rel = IntMap Team
type Val = IntMap (Set Proposition)

data KrM = KrM {worlds :: Team,
                rel    :: Rel,
                val    :: Val}
  deriving (Show)
```

We also define the following shorthands for looking up successors and valuations of worlds, that allow us to treat `rel` and `val` as (partial) functions:

```
rel' :: KrM -> World -> Team
rel' = (IntMap.!) . rel

val' :: KrM -> World -> Set Proposition
val' = (IntMap.!) . val
```

To allow defining a (anti)support-relation for different structures as well, we use the following typeclasses, that contain a both a curried and uncurried version of the function for (anti)support for ease of use (only one is required to be provided; the other is the curried/uncurried equivalent).

```
class Supportable model state formula where
  support :: model -> state -> formula -> Bool
  support = curry (|=)

  (|=) :: (model, state) -> formula -> Bool
  (|=) = uncurry support

  {-# MINIMAL (|=) | support #-}

class Antisupportable model state formula where
  antisupport :: model -> state -> formula -> Bool
  antisupport = curry (=|)

  (=|) :: (model, state) -> formula -> Bool
  (=|) = uncurry antisupport

  {-# MINIMAL (=|) | antisupport #-}
```

Before we implement the semantics, we need a function that computes all pairs of teams whose union is a given team s. Naively, we may define e.g.

```
teamParts :: Team -> [(Team, Team)]
teamParts s = [(t,u) | t <- ps, u <- ps, sort . nub (t ++ u) == sort s]
  where ps = subsequences s
```

Computationally however, this is incredibly expensive and will form a major bottleneck for the efficiency of the model checking. While finding such partitions is inherently exponential in complexity, we can still do slightly better (at least on average) than the above:

```
teamParts :: Team -> [(Team, Team)]
teamParts s = do
    t <- subsequences s
    u <- subsequences t
    return (t, s \\ u)
```

Now, we are ready to define our semantics, in accordance to [?]:

```
instance Supportable KrM Team Form where
  (_,s) |= Bot     = null s
```

```
    (_,s) |= NE      = not (null s)
    (m,s) |= Prop n  = all (elem n . val' m) s
    (m,s) |= Neg f   = (m,s) =| f
    (m,s) |= And f g = (m,s) |= f && (m,s) |= g
    (m,s) |= Or f g  = any (\(t,u) -> (m,t) |= f && (m,u) |= g) $ teamParts s
    (m,s) |= Gor f g = (m,s) |= f || (m,s) |= g
    (m,s) |= Dia f   = all (any (\t -> not (null t) && (m,t) |= f) . subsequences . rel' m) s

instance Antisupportable KrM Team Form where
    _       =| Bot    = True
    (_,s) =| NE       = null s
    (m,s) =| Prop n   = not $ any (elem n . val' m) s
    (m,s) =| Neg f    = (m,s) |= f
    (m,s) =| And f g  = any (\(t,u) -> (m,t) =| f && (m,u) =| g) $ teamParts s
    (m,s) =| Or f g   = (m,s) =| f && (m,s) =| g
    (m,s) =| Gor f g  = (m,s) =| Or f g
    (m,s) =| Dia f    = all (\w -> (m, rel' m w) =| f) s
```

One may also easily extend the above semantics to lists of formulae, as shown below.

```
instance Supportable KrM Team [Form] where
  support = (all .) . support

instance Antisupportable KrM Team [Form] where
  antisupport = (all .) . antisupport
```

1.3   Random models

As for formulas, we want to be able to generate random models to verify properties of BSML, and will use QuickCheck for this.

We will need a function that generates a `Set` containing random elements of a list, so we define the following analogue of `sublistOf`:

```
subsetOf :: Ord a => [a] -> Gen (Set a)
subsetOf = (Set.fromList <$>) . sublistOf
```

When shrinking the valuation of models, we want to have QuickCheck try valuation with fewer propositions occurring in the model, but it does not make sense to shrink the values of the propositions, so we define the following functions to only shrink which propositions occur:

```
-- Shrink a list without shrinking individual elements
shrinkList' :: [a] -> [[a]]
shrinkList' = shrinkList (const [])

-- Shrink a set without shrinking individual values
shrinkSet :: Set Proposition -> [Set Proposition]
shrinkSet = fmap Set.fromList . shrinkList' . Set.toList

-- Shrink a valuation by uniformly restricting the occuring propositions
shrinkVal :: Val -> [Val]
shrinkVal v = do
  let allProps = Set.unions v
  newProps <- shrinkSet allProps
  return (Set.intersection newProps <$> v)
```

Then, an arbitrary model $M = (W, R, V)$ can be generated as follows:

```
instance Arbitrary KrM where
  arbitrary = sized $ \n -> do
```

The size parameter n gives an upper bound to the amount of worlds; we pick some $k \leq n$ and define $W = 0, 1, \ldots, k$. For every $w <= k$, we then generate a random set of successors $R[w]$ and random set of propositions $V(w)$ that hold at w.

```
k <- choose (0, n)
let ws = [0..k]
r <- IntMap.fromList . zip ws <$> vectorOf (k+1) (sublistOf [0..k])
v <- IntMap.fromList . zip ws <$> vectorOf (k+1) (subsetOf [1..numProps])
return (KrM ws r v)
```

When finding counterexamples, it is useful to find models that are as small as possible, so we also define `shrink` that tries to restrict the worlds of the model.

```
shrink m = do
  ws' <- init $ subsequences $ worlds m
  let r' = IntMap.fromList [(w, rel' m w `intersect` ws') | w <- ws']
  let v' = IntMap.filterWithKey (const . (`elem` ws')) $ val m
  KrM ws' r' <$> shrinkVal v'
```

When testing, we will often want to generate a random model *with* a random team or world of that model. Generating a random `Int` or `[Int]` would not work then, since there is no guarantee that the generated value is a valid team/world in the model. To remedy that, we define wrappers for models with a team/world and define `Arbitrary`-instances for those wrappers:

```
data TeamPointedModel = TPM KrM Team
  deriving (Show)

data WorldPointedModel = WPM KrM World
  deriving (Show)

instance Arbitrary TeamPointedModel where
  arbitrary = do
    m <- arbitrary
    s <- sublistOf $ worlds m
    return (TPM m s)

  shrink (TPM m s) = filter (\(TPM m' s') -> s' `isSubsequenceOf` worlds m') (TPM <$>
      shrink m <*> shrinkList' s)

instance Arbitrary WorldPointedModel where
  arbitrary = do
    m <- arbitrary
    w <- elements $ worlds m
    return (WPM m w)

  shrink (WPM m w) = filter (\(WPM m' w') -> w' `elem` worlds m') (WPM <$> shrink m <*> [w
      ])
```

Note that when shrinking, we should only allow shrinks where the world/team is still contained in the model.

1.4   Modal Logic

To define the *pragmatic enrichment* function mentioned in the introduction, we define a type to represent formulas of basic Modal Logic (ML) and implement the standard Kripke semantics.

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE LambdaCase #-}
module ML where

import Syntax
import Semantics
```

```
import Test.QuickCheck
```

```
-- Basic Modal Logic formulas
data MForm
  = MProp Proposition
  | MNeg MForm
  | MAnd MForm MForm
  | MOr  MForm MForm
  | MDia MForm
  deriving (Eq,Show)

-- Kripke semantics
instance Supportable KrM World MForm where
  (m,w) |= MProp n  = n 'elem' val' m w
  (m,w) |= MNeg f   = not $ (m,w) |= f
  (m,w) |= MAnd f g = (m,w) |= f && (m,w) |= g
  (m,w) |= MOr f g  = (m,w) |= f || (m,w) |= g
  (m,w) |= MDia f   = any (\v -> (m,v) |= f) $ rel' m w
```

Formally, pragmatic enrichment is given by the function $[\cdot]^+ : \mathrm{ML} \to \mathrm{BSML}$, defined as

$$[p]^+ := p \wedge \mathtt{NE} \qquad\qquad \text{for } p \in \mathtt{Prop}$$
$$[\heartsuit\varphi]^+ := \heartsuit[\varphi]^+ \wedge \mathtt{NE} \qquad\qquad \text{for } \heartsuit \in \{\neg, \Diamond\}$$
$$[\varphi \odot \psi]^+ := ([\varphi]^+ \odot [\psi]^+) \wedge \mathtt{NE} \qquad\qquad \text{for } \odot \in \{\vee, \wedge\}$$

which is straightforward to implement in Haskell.

```
-- The pragmatic enrichment function [.]+ : ML -> BSML.
enrich :: MForm -> Form
enrich (MProp n) = Prop n 'And' NE
enrich (MNeg f)  = Neg (enrich f) 'And' NE
enrich (MDia f)  = Dia (enrich f) 'And' NE
enrich (MAnd f g) = (enrich f 'And' enrich g) 'And' NE
enrich (MOr f g)  = (enrich f 'Or'  enrich g) 'And' NE
```

To test the semantic effect of enrichment, and some other properties of ML as a fragment of BSML, we also implement the canonical embedding of ML into BSML.

```
-- Embedding ML >-> BSML
toBSML :: MForm -> Form
toBSML (MProp n) = Prop n
toBSML (MNeg f)  = Neg (toBSML f)
toBSML (MAnd f g) = And (toBSML f) (toBSML g)
toBSML (MOr f g)  = Or (toBSML f) (toBSML g)
toBSML (MDia f)  = Dia (toBSML f)
```

As the reader should expect at this point, we also implement an arbitrary instance for formulas of ML, which is completely analogous to that for BSML:

```
randomMProp :: Gen MForm
randomMProp = MProp <$> choose (1, numProps)

instance Arbitrary MForm where
  arbitrary = sized $ \case
    0 -> randomMProp
    _ -> oneof [
      randomMProp,
      MNeg <$> f,
      MAnd <$> f <*> f,
      MOr  <$> f <*> f,
      MDia <$> f]
    where f = scale ('div' 2) arbitrary

  shrink (MNeg f)    = f        : [MNeg f'     | f'          <- shrink f]
  shrink (MDia f)    = f        : [MDia f'     | f'          <- shrink f]
  shrink (MAnd f1 f2) = [f1, f2] ++ [MAnd f1' f2' | (f1',f2') <- shrink (f1,f2)]
  shrink (MOr f1 f2)  = [f1, f2] ++ [MOr  f1' f2' | (f1',f2') <- shrink (f1,f2)]
```

```
    shrink _              = []
```

2 Lexer

Below we describe a brief code for a lexer we built for BSML formulae. The lexer was built to utilise the alex lexing tool mentioned in class.

We note that this code is heavily based on the code of Katejan Dvoracek, who built a Natural Deduction Prover, and used within it a lexer running on Alex, and a parser running on Happy. You can find the PDF for the project here. Thanks a bunch Katejan!

```
{
module Lexer where
import Token
}


%wrapper "posn"
$dig = 0 -9 -- digits
tokens:-
-- ignore whitespace and comments :
$white + ;
" --" .* ;
-- keywords and punctuation :
"(" { \ p _ -> TokenOB p }
")" { \ p _ -> TokenCB p }
"_|_" { \ p _ -> TokenBot p }
"NE" {\ p _ -> TokenNE p}
"[]" { \ p _ -> TokenBox p }
"<>" { \ p _ -> TokenDmd p }
"~" { \ p _ -> TokenNot p }
"&" { \ p _ -> TokenCon p }
"v" { \ p _ -> TokenDis p }
-- Integers and Strings :
$dig + { \ p s -> TokenInt ( read s ) p }
[P - Z ] { \ p s -> TokenPrp ( ord ( head s ) - ord 'P') p }
[p - z ] { \ p s -> TokenPrp ( ord ( head s ) - ord 'p' + 11) p }
```

The basic overview of the file above is as follows: We instruct Alex to construct a lexer that goes through a given string and picks out "tokens" from it. These tokens are symbols to be kept in mind while the string is being assigned meaning - which is done using a parser built by Happy.

This particular file has a .x extension - once written, the user may run alex Lexer.x in their terminal. This command generates Lexer.hs, a fully functioning lexer for the above tokens built by Alex. The code is housed inside a module called Lexer, which we import within the parser file.

3 Parser

We now take a look at the code for the parser, which works using Happy. As with the lexer, the code is inspired by Katejan Dvoracek's work for a lexer and parser for Natural Deductions.

We first import the modules that we require to run the parser.

```
> {
> module Parser where
> import Data.Char
> import Token
> import Lexer
> import Syntax
```

```
> import Semantics
> import ML
> }
```

Below, we describe the tokens that Happy needs to keep track of while reading the string. These tokens were described in the module Lexer.

```
> %name formula
> %tokentype { Token AlexPosn }
> %monad { ParseResult }
>
> %token
> BOT { TokenBot _ }
> NE {TokenNE _}
> AND { TokenCon _ }
> OR { TokenDis _ }
> NOT { TokenNot _ }
> BOX { TokenBox _ }
> DMD { TokenDmd _ }
> GDIS {TokenGDIS _}
> NUM { TokenInt $$ _ }
> '(' { TokenOB _ }
> ')' { TokenCB _}
```

We describe also the binding hierarchy for binary operators in our language. The order of precedence is described by listing operators from weakest to strongest, as evidenced below. Note that the binding for all unary operators is stronger than the binding for binary operators, and unary operators operate at the same binding strength. This behaviour keeps in line with the way we would like to parse formulae in our language.

```
> %left OR
> %left AND
> %left GDIS
>
> %%
```

We now detail two different types of formulae - bracketed formulae and non-bracketed formulae. The reasoning for the distinction is rather simple - the presence of parentheses around a formula requires that any operations within the parentheses need to be given priority over operations outside of them. This may break regular precedence rules, and hence need to be accounted for.

```
> Form :: { Form }
> Form : BrForm { $1 }
> | Form AND Form { And $1 $3 }
> | Form OR Form { Or $1 $3 }
> | Form GDIS Form {Gor $1 $3}
> BrForm :: { Form }
> BrForm : NUM { Prop $1 }
> | BOT { Bot }
> | NE { NE }
> | NOT BrForm { Neg $2 }
> | BOX BrForm { Syntax.box $2 }
> | DMD BrForm { Dia $2 }
> | '(' Form AND Form ')' { And $2 $4 }
> | '(' Form OR Form ')' { Or $2 $4 }
> | '(' Form GDIS Form ')' { Gor $2 $4}
```

Next, we outline error messages for our parser. These error messages (very helpfully and wonderfully described by Katejan) describe where the error occurs exactly in the string, and why Happy failed to parse it.

```
> {
> data ParseError = ParseError { pe_str :: String
>                              ,pe_msg :: String
```

```
>                                       ,pe_col :: Int}
>       deriving (Eq , Show)
>
> type ParseResult a = Either ParseError a
>
>
> happyError :: [ Token AlexPosn ] -> ParseResult a
> happyError [] = Left $
>       ParseError { pe_str = " " , pe_msg = " Unexpected end of input : " , pe_col = -1}
> happyError ( t : ts ) = Left $
>       ParseError { pe_str = " " , pe_msg = " Parse error : " , pe_col = col }
>       where ( AlexPn abs lin col ) = apn t
>
> myAlexScan :: String -> ParseResult [ Token AlexPosn ]
> myAlexScan str = go ( alexStartPos , '\n' ,[] , str )
>    where
>        go :: AlexInput -> ParseResult [ Token AlexPosn ]
>        go inp@( pos ,_ ,_ , str ) =
>            case alexScan inp 0 of
>                AlexEOF -> Right []
>                AlexError (( AlexPn _ _ column ) ,_ ,_ , _ ) -> Left $
>                    ParseError { pe_str = str , pe_msg = " Lexical error : " , pe_col =
>    column -1}
>                AlexSkip inp' len -> go inp'
>                AlexToken inp' len act -> go inp'  >>=
>                    (\ x -> Right $ act pos ( take len str ) : x )
```

Finally, we describe the actual parsing function itself, called `parseFormula`. Upon running `happy Parser.ly`, we get a Haskell file `Parser.hs` which contains the parseFormula function. The output for parseFormula is of type `Either ParseError Form`, so we know that the user has an appropriate input when they receive an output prefixed by the constructor `Left`.

```
> parseFormula :: String -> ParseResult Form
> parseFormula str = go $ myAlexScan str  >>= formula
>    where
>        go ( Left err ) = Left $ err { pe_str = str }
>        go ( Right res ) = Right res
>
> }
```

4   Simple Tests

We now use the library QuickCheck to randomly generate input for our functions and test some properties.

```
module Main where

import Test.QuickCheck
import Test.Hspec
import Test.Hspec.QuickCheck

import Semantics
import Parser
import Syntax
import ML
import Models ( m3b, m3c, mM, s3b, s3c, sMA, sMB, sMC, sMD )
```

The following uses the HSpec library to define different tests. We use a mix of QuickCheck and specific inputs, depending on what we are testing for.

The "Figure 3" section corresponds to the three examples labeled 3a, 3b, and 3c [**?**]. The paper gives a couple formulas per example to illustrate the semantics of BSML. We test each of these formulas to confirm our implementation contains the expected semantics.

```haskell
main :: IO ()
main = hspec $ do
  describe "Figure 3" $ do
    it "Figure 3b, [<>(p ^ q)]+" $
      (m3b, s3b) |= enrich (MDia (ma `MAnd` mb)) `shouldBe` False
    it "Figure 3c, [<>(p v q)]+" $
      (m3c, s3c) |= enrich (MDia (ma `MOr` mb)) `shouldBe` True

  describe "Motivating Example" $ do
    it "(a) |= (a v b) == True" $
      (mM, sMA) |= toBSML (ma `MOr` mb) `shouldBe` True
    it "(a) |= [a v b]+  == True" $
      (mM, sMA) |= enrich (ma `MOr` mb) `shouldBe` True

    it "(b) |= (a v b) == True" $
      (mM, sMB) |= toBSML (ma `MOr` mb) `shouldBe` True
    it "(b) |= [a v b]+ == True" $
      (mM, sMB) |= enrich (ma `MOr` mb) `shouldBe` True

    it "(c) |= (a v b) == True" $
      (mM, sMC) |= toBSML (ma `MOr` mb) `shouldBe` True
    it "(c) |= [a v b]+ == False" $
      (mM, sMC) |= enrich (ma `MOr` mb) `shouldBe` False

    it "(d) |= (a v b) == False" $
      (mM, sMD) |= toBSML (ma `MOr` mb) `shouldBe` False
    it "(d) |= [a v b]+  == False" $
      (mM, sMD) |= enrich (ma `MOr` mb) `shouldBe` False

    prop "Arbitrary |= (a v b) !== Arbitrary |= [a v b]+  == False" $
      expectFailure $ \(TPM m s) -> (m,s) |= toBSML (ma `MOr` mb) == (m,s) |= enrich (ma `
          MOr` mb)
```

We will expand this section later, by adding more tautologies that should hold for BSML logic ensuring our implementation is correct. Here we use QuickCheck, but we need to limit the maximal size of the arbitrary models we generate. This is necessary because the evaluation of support in team semantics is inherently exponential in complexity (see e.g. the clause for support of disjunctions).

The paper [?] discusses various interesting properties that should hold for our implementation. Narrow-scope and wide-scope relate to the "pragmatic enrichment function." The flatness test confirms that our implementation of ML formulas are flat.

```haskell
  describe "Properties from Paper" $ modifyMaxSize (const 25) $ do
    prop "NarrowScope, <>(a v b) =| (<>a ^ <>b)" $
      \(TPM m s) -> (m,s) |= enrich (MDia (ma `MOr` mb)) == (m,s) |= enrich (MDia ma `MAnd`
          MDia mb)
    prop "Wide Scope, <>a v <>b) =| <>a ^ <>b" $
      \(TPM m s) -> all (\w -> rel' m w == s) s <= ((m,s)  |= enrich (MDia ma `MOr` MDia mb
          ) <= (m,s) |= enrich (MDia ma `MAnd` MDia mb))

  describe "Flatness" $ modifyMaxSize (const 15) $ do
    modifyMaxSize (const 10) $ prop "(M,s) |= f <==> M,{w} |= f forall w in s" $
      \(TPM m s) f -> (m,s) |= toBSML (f::MForm) == all (\w -> (m, [w]) |= toBSML f) s
    prop "M,{w} |= f <==> M,w |= f" $
      \(WPM m w) f -> (m, [w]) |= toBSML (f::MForm) == (m,w) |= f
    prop "Full BSML is *not* flat" $ expectFailure $
      \(TPM m s) f -> (m,s) |= (f::Form) == all (\w -> (m, [w]) |= f) s

  describe "Pretty Print and Parsing" $ do
    prop "formula -> prettyPrint -> Parsing == original formula" $
      \f -> parseFormula (ppForm (f::Form)) == Right f


  where
    ma = MProp 1
    mb = MProp 2
```

14

To run the tests, use `stack test`.

To also find out which part of your program is actually used for these tests, run `stack clean && stack test --cove`
Then look for "The coverage report for ... is available at ... .html" and open this file in your browser.
See also: `https://wiki.haskell.org/Haskell_program_coverage`.

5   Conclusion