

Interpreting Music and Dance in Haskell

Jennifer Streb and *Perry Alexander*
The University of Kansas - ITTC
2335 Irving Hill Rd, Lawrence, KS 66045
{jenis,alex}@ittc.ku.edu

October 29, 2004

Abstract

This file is the second attempt at the dance language. In this version Algebras and Functors are used to define such functions as eval and typeof. Monads are also used in this version. Specifically the Reader and Error monads. The Reader monad is used to pass around the environment which is located in Environment file. The Error monad is used in the definition of evaluation.

1 Language Utilities

Definitions:

- a - Value space - *Integer*
- $f\ a$ - Language defined over a value space
- $map_f :: (a \rightarrow b) \rightarrow (f\ a) \rightarrow (f\ b)$ - Function provided by a *Functor* that transforms one language generated by the constructors for type f into another language generated by the constructors for type f .
- $\phi :: f\ a \rightarrow a$ - Function provided by an *Algebra* that transforms a language element into a value element.

The *LangUtils* module provides classes, data structures and functions for developing the Jete type language and movement language.

```

{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances -fallow-overlapping-instances #-}
module LangUtils where

import Control.Monad
import Control.Monad.Reader
import Control.Monad.Error
import Data.Maybe
import Data.List

```

An actual *Movement* will be one of the above movements or a movement followed by a movement or a movement in parallel with another movement. We'll use a *Sum* type to put the movements together.

First define the *Sum* data type:

```

data Sum f g x = S (Either (f x) (g x))
      unS (S x) = x

```

Note that the *Sum* type differs from the standard *Either* type with the presence of a third parameter.

To understand the *Functor* and *Algebra* classes with respect to languages it is important to remember that in our definitions *a* is a value space and *f a* is a term language defined over that value space. For example, if *a* is *Integer* and *f* is *List*, then *f a* is the language of lists defined over integers. When we define functors, we are defining a transform, map_f from a language defined over one value space to the same language defined over another value space. When we define algebras, we are defining a transform, ϕ , from a language defined over a value space to the value space itself. In effect, ϕ provides an evaluation function.

It is important to remember the definition of *Functor* provide in the standard Prelude:

```

class Functor f where
      map_f :: (a → b) → f a → f b

```

A *Functor*, *f*, is some constructor that we can define a map, map_f , over. The signature of map_f defines a function that given a function mapping type *a* to type *b* and an instance of the functor, *f a*, will generate an instances

of the functor over b , $f b$. For example, f is a *List* the map_f is the built in *map* function. In our case, f is an abstract syntax tree representing the type language or expression language that define our overall language.

$a \quad {}^F F(a)$

$b \quad {}^F F(b)$

We define *Algebra* and *Coalgebra* classes that define ϕ and ψ respectively. If f is a functor, then $f a$ is an *Algebra* if ϕ can be defined mapping an application of f to it's argument type a . Conversely, the *CoAlgebra* defines a mapping ψ from the argument type to a *Functor* application.

```
class Functor f  $\Rightarrow$  Algebra f a where
   $\phi :: f a \rightarrow a$ 
```

```
class Functor f  $\Rightarrow$  Coalgebra f a where
   $\psi :: a \rightarrow f a$ 
```

The *Rec* data type defines a fixed point type for some type f having one argument. In effect, the fixed point type creates a parameter-less type from the single parameter type. The constructor *In* is necessary to define a data type.

```
data Rec f = In (f (Rec f))
  out :: Rec f  $\rightarrow$  f (Rec f)
  out (In x) = x
```

The *cata* function represents a catamorphism, or fold, over the recursive structure. The prerequisite *Algebra f a* assures the existance of $\phi a \rightarrow a$. $map_f \text{ cata}$ applies the *cata* function across the data structure involved in the functor. If *Algebra f a* holds, then we also know that *Functor f* holds and map_f exists. Thus, $map_f \text{ cata}$ thus maps *cata* over *out* extracts the argument to *Rec f*.

```
 $cata :: (Algebra f a) \Rightarrow Rec f \rightarrow a$ 
 $cata = \phi \circ map_f \text{ cata} \circ out$ 
```

If f and g are functors, then *Sum f g* is a functor where map_f operates on the left and right parts of the sum.

```

instance (Functor f, Functor g) => Functor (Sum f g) where
  map_f h (S (Prelude.Left x)) = S (Prelude.Left (map_f h x))
  map_f h (S (Prelude.Right x)) = S (Prelude.Right (map_f h x))

```

If $f a$ and $g a$ are algebras, then $Sum f g$ is also an algebra where ϕ is the sum of the ϕ functions from the original algebra. Use the built-in *either* function to select the appropriate ϕ . Also need to *unS* the result to remove the *S* constructor.

```

instance (Algebra f a, Algebra g a) => Algebra (Sum f g) a
  where  $\phi = \text{either } \phi \phi \circ \text{unS}$ 

```

Define *Subtype* in the classical way by defining \uparrow and \downarrow using the *Maybe* type for \downarrow .

```

class Subtype a b where
   $\uparrow :: a \rightarrow b$ 
   $\downarrow :: b \rightarrow \text{Maybe } a$ 

```

Every type x is a subtype of itself where $\uparrow = id$ and $\downarrow = Just$.

```

instance Subtype x x where
   $\uparrow = id$ 
   $\downarrow = Just$ 

```

```

mPrj :: (Error e, (MonadError e m), Show b, Subtype a b) => b -> m a
mPrj x = maybe (throwError $ strMsg errMsg) return ( $\downarrow$  x)

```

```

where
  errMsg = "Type error: Cannot project '" ++
    (show x) ++
    "' to the desired type"

```

```

retInj x = return ( $\uparrow$  x)

```

Subsum is the magic class that makes all this go. It's like *Subtype* except that f and g are parameterized.

```

class Subsum f g where
  ↑S :: f x → g x
  ↓S :: g x → Maybe (f x)

```

Every f is a *Subsum* of itself.

```

instance Subsum f f where
  ↑S f = f
  ↓S = Just

```

Any f is a *Subsum* of *Sum f g* where $S \circ \uparrow_S$ is the *Left* constructor.

```

instance Subsum f (Sum f g) where
  ↑S = S ∘ Prelude.Left
  ↓S (S (Prelude.Left f)) = Just f
  ↓S (S (Prelude.Right f)) = Nothing

```

If *Subsum f g* holds, then *Subsum f (Sum x g)* holds where $S \circ \text{Right}$ is the injection function. The projection function is interesting because of the way *Sum* is used to represent abstract syntax. Here we project from the *Right* value because.

```

instance (Subsum f g) ⇒ Subsum f (Sum x g) where
  ↑S = S ∘ Prelude.Right ∘ ↑S
  ↓S (S (Prelude.Left x)) = Nothing
  ↓S (S (Prelude.Right b)) = ↓S b

```

```

toSum :: (Subsum f g, Functor g) ⇒ f (Rec g) → Rec g
toSum x = In (↑S x)

```

2 Dance AST Definitions

The *DanceAST* module provides data structures and functions for manipulating representations for movement.

```
{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances -fallow-overlapping-instances #-}
```

module *Dance2AST* **where**

import *Ratio*
import *Control.Monad*
import *Control.Monad.Reader*
import *Control.Monad.Error*
import *Data.Maybe*
import *Data.List*
import *LangUtils*

—————-Creating Sum—————

type *MovementS* = (*Sum TurnPrep*
 (*Sum TurnMiddle*
 (*Sum LeapPrep*
 (*Sum LeapMiddle*
 (*Sum End*
 (*Sum BattementBegEnd*
 (*Sum BattementMiddle*
 (*Sum RondDeJambe*
 (*Sum Filler*
 (*Sum VarTerm*
 (*Sum FcnTerm*
 (*Sum Continuation MovementType*))))))))))

type *MovementLang* = *Rec MovementS*

instance *Show (Rec MovementS)* **where**
 show x = show (out x)

instance ((*Show (f x)*), (*Show (g x)*)) \Rightarrow *Show (Sum f g x)* **where**
 show (S (Prelude.Left x)) = show x
 show (S (Prelude.Right x)) = show x

—————-toSum function—————

toMovementLang :: (*Subsum f MovementS*) \Rightarrow *f MovementLang* \rightarrow *MovementLang*
toMovementLang = toSum

data *VarTerm* *m* = *Var String* **deriving** (*Show*, *Eq*)

instance *Functor VarTerm* **where**

map_f *f* (*Var s*) = *Var s*

data *FcnTerm* *m* = *Lambda [String] m*
| *App m [m]*
deriving (*Show*, *Eq*)

instance *Functor FcnTerm* **where**

map_f *f* (*Lambda vt x*) = (*Lambda vt (f x)*)

map_f *f* (*App a1 a2*) = *App (f a1) (map f a2)*

data *Ty_D* = *TurnPrepTy TPrepType*
| *TurnMiddleTy TurnType*
| *LeapPrepTy LPrepType*
| *LeapMiddleTy LeapType*
| *LandingTy LandingType*
| *KickPrepLandTy KPrepType*
| *KickMiddleTy KickType*
| *RondeDeJambeTy RondDeJambeType*
| *FillerTy FillerType*
| *ContinuationTy*
| *TurnEndTy TurnType LandingType*
| *TurnTy TPrepType TurnType LandingType*
| *LeapEndTy LeapType LandingType*
| *LeapTy LPrepType LeapType LandingType*
| *KickEndTy KickType KPrepType*
| *KickTy KPrepType KickType KPrepType*
deriving (*Eq*, *Show*)

data *FootPositionTy* = *FirstFTy*
| *SecondFTy*
| *ThirdFTy*
| *FourthFTy*
| *FifthFTy*
deriving (*Eq*, *Show*)

data *TPrepType* = *PrepareType Direction*
| *StepPrepareType Direction*
deriving (*Eq, Show*)

data *TurnType* = *PirouetteType LegPosition*
| *FouetteType LegPosition*
| *ChaineType LegPosition*
| *PiqueType LegPosition*
deriving (*Eq, Show*)

data *LPrepType* = *ChasseType Direction*
| *StepStepType Direction*
deriving (*Eq, Show*)

data *LeapType* = *JeteType Direction*
| *SwitchType Direction*
| *CenterLeapType Direction*
deriving (*Eq, Show*)

data *LandingType* = *LandType FootPosition*
deriving (*Eq, Show*)

data *KPrepType* = *StepType Direction*
deriving (*Eq, Show*)

data *KickType* = *GrandeType Direction*
| *PetitType Direction*
deriving (*Eq, Show*)

data *RondDeJambeType* = *RondDeJambeType Direction*
deriving (*Eq, Show*)

data *FillerType* = *WalkType*
| *HoldType*
| *ContinueType*
deriving (*Eq, Show*)

```
data Direction = Left
                | Right
                | Center
                deriving (Eq, Show)
```

```
instance Functor Direction where
  mapf f (Dance2AST.Left) = Dance2AST.Left
  mapf f (Dance2AST.Right) = Dance2AST.Right
  mapf f (Center) = Center
```

```
data FootPosition = FirstF
                    | SecondF
                    | ThirdF
                    | FourthF
                    | FifthF
                    | PrevF
                    | NextF
                    deriving (Eq, Show)
```

```
instance Functor FootPosition where
  mapf f (FirstF) = FirstF
  mapf f (SecondF) = SecondF
  mapf f (ThirdF) = ThirdF
  mapf f (FourthF) = FourthF
  mapf f (FifthF) = FifthF
  mapf f (PrevF) = PrevF
  mapf f (NextF) = NextF
```

```
data ArmPosition = FirstA
                    | SecondA
                    | ThirdA
                    | FourthA
                    | FifthA
                    | HighV
                    | LowV
                    | Tee
                    | RDiagonal
                    | LDiagonal
                    | PrevA
```

| *NextA*
deriving (*Eq, Show*)

instance *Functor ArmPosition* **where**

map_f *f* (*FirstA*) = *FirstA*
map_f *f* (*SecondA*) = *SecondA*
map_f *f* (*ThirdA*) = *ThirdA*
map_f *f* (*FourthA*) = *FourthA*
map_f *f* (*FifthA*) = *FifthA*
map_f *f* (*HighV*) = *HighV*
map_f *f* (*LowV*) = *LowV*
map_f *f* (*Tee*) = *Tee*
map_f *f* (*RDiagonal*) = *RDiagonal*
map_f *f* (*LDiagonal*) = *LDiagonal*
map_f *f* (*PrevA*) = *PrevA*
map_f *f* (*NextA*) = *NextA*

data *LegPosition* = *Attitude*
| *Coupe*
| *Passe*
| *OpenPasse*
| *Develope*
| *Straight*
deriving (*Eq, Show*)

instance *Functor LegPosition* **where**

map_f *f* (*Attitude*) = *Attitude*
map_f *f* (*Coupe*) = *Coupe*
map_f *f* (*Passe*) = *Passe*
map_f *f* (*OpenPasse*) = *OpenPasse*
map_f *f* (*Develope*) = *Develope*
map_f *f* (*Straight*) = *Straight*

Turn Preparations

data *TurnPrep m* = *Prepare Direction FootPosition ArmPosition*
| *StepPrepare Direction FootPosition ArmPosition FootPosition ArmPosition*
deriving (*Eq, Show*)

----- Functor -----

instance Functor TurnPrep where

$\text{map}_f f (\text{Prepare } d \text{ fp } ap) = \text{Prepare } d \text{ fp } ap$

$\text{map}_f f (\text{StepPrepare } d \text{ fp1 } ap1 \text{ fp2 } ap2) = \text{StepPrepare } d \text{ fp1 } ap1 \text{ fp2 } ap2$

----- Turns -----

data TurnMiddle m = Pirouette LegPosition ArmPosition

| Fouette LegPosition ArmPosition

| Chaine LegPosition ArmPosition

| Pique LegPosition ArmPosition

deriving (Eq, Show)

----- Functor -----

instance Functor TurnMiddle where

$\text{map}_f f (\text{Pirouette } lp \text{ ap}) = \text{Pirouette } lp \text{ ap}$

$\text{map}_f f (\text{Fouette } lp \text{ ap}) = \text{Fouette } lp \text{ ap}$

$\text{map}_f f (\text{Chaine } lp \text{ ap}) = \text{Chaine } lp \text{ ap}$

$\text{map}_f f (\text{Pique } lp \text{ ap}) = \text{Pique } lp \text{ ap}$

----- Leap Prep -----

data LeapPrep m = Chasse Direction

| StepStep Direction

deriving (Eq, Show)

----- Functor -----

instance Functor LeapPrep where

$\text{map}_f f (\text{Chasse } d) = \text{Chasse } d$

$\text{map}_f f (\text{StepStep } d) = \text{StepStep } d$

----- Leaps -----

```
data LeapMiddle m = Jete Direction LegPosition ArmPosition
                  | Switch Direction LegPosition ArmPosition
                  | CenterLeap Direction LegPosition ArmPosition
deriving (Eq, Show)
```

————— Functor —————

```
instance Functor LeapMiddle where
  mapf f (Jete d lp ap) = Jete d lp ap
  mapf f (Switch d lp ap) = Switch d lp ap
  mapf f (CenterLeap d lp ap) = CenterLeap d lp ap
```

————— Land —————

```
data End m = Land FootPosition ArmPosition
deriving (Eq, Show)
```

————— Functor —————

```
instance Functor End where
  mapf f (Land fp ap) = Land fp ap
```

————— Battement Prep/End —————

```
data BattementBegEnd m = Step Direction
deriving (Eq, Show)
```

————— Functor —————

```
instance Functor BattementBegEnd where
  mapf f (Step d) = Step d
```

————— Battement Middle —————

```
data BattementMiddle m = Grande Direction LegPosition ArmPosition
                       | Petit Direction LegPosition ArmPosition
deriving (Eq, Show)
```

----- Functor -----

instance *Functor BattementMiddle* **where**
 $map_f f (Grande\ d\ lp\ ap) = Grande\ d\ lp\ ap$
 $map_f f (Petit\ d\ lp\ ap) = Petit\ d\ lp\ ap$

----- Rond De Jambe -----

data *RondDeJambe* $m = RondDeJambe\ Direction\ ArmPosition\ FootPosition\ ArmPosition$
 deriving (*Eq, Show*)

----- Functor -----

instance *Functor RondDeJambe* **where**
 $map_f f (RondDeJambe\ d\ ap1\ fp\ ap2) = RondDeJambe\ d\ ap1\ fp\ ap2$

----- Fillers -----

data *Filler* $m = Walk$
 | *Hold*
 | *Continue*
 deriving (*Eq, Show*)

----- Functor -----

instance *Functor Filler* **where**
 $map_f f (Walk) = Walk$
 $map_f f (Hold) = Hold$
 $map_f f (Continue) = Continue$

----- Follows and While -----

data *Continuation* $m = Follows\ m\ m$
 | *While\ m\ m*
 deriving (*Eq, Show*)

----- Functor -----

instance *Functor Continuation* **where**

map_f *f* (*Follows* *m*₁ *m*₂) = (*Follows* (*f* *m*₁) (*f* *m*₂))

map_f *f* (*While* *m*₁ *m*₂) = (*While* (*f* *m*₁) (*f* *m*₂))

–Parser Data Structures–

data *Routine* *m* = *Routine* *RNRet* *StyleRet* *Integer* *DescripRet* [(*EightCount* *m*)] *DescripR*
deriving *Show*

data *StyleRet*
= *Jazz*
| *HipHop*
| *Lyrical*
| *Novelty*
| *Pom*
| *Prop*
deriving (*Show*, *Eq*)

data *MCountRet* =
MCDash *Integer* *Integer* |
MCComma *Integer* *Integer* |
MCIntAnd *Integer* |
MCAnd |
MCInt *Integer*
deriving (*Show*, *Eq*)

data *MovementType* *m* = *DirectionReturn* *Direction*
| *FootPosReturn* *FootPosition*
| *ArmPosReturn* *ArmPosition*
| *LegPosReturn* *LegPosition*
| *TurnPrepReturn* (*TurnPrep* *m*)

```

| TurnMiddleReturn (TurnMiddle m)
| LeapPrepReturn (LeapPrep m)
| LeapMiddleReturn (LeapMiddle m)
| EndReturn (End m)
| BattementBegEndReturn (BattementBegEnd m)
| BattementMiddleReturn (BattementMiddle m)
| RondDeJambeReturn (RondDeJambe m)
| FillerReturn (Filler m)
deriving (Eq, Show)

```

instance *Functor MovementType* **where**

```

map_f f (DirectionReturn d) = DirectionReturn d
map_f f (FootPosReturn fp) = FootPosReturn fp
map_f f (ArmPosReturn ap) = ArmPosReturn ap
map_f f (LegPosReturn lp) = LegPosReturn lp
map_f f (TurnPrepReturn (Prepare d lp ap)) = (TurnPrepReturn (Prepare d lp ap))
map_f f (TurnPrepReturn (StepPrepare d lp1 ap1 lp2 ap2)) = (TurnPrepReturn (StepPr
map_f f (TurnMiddleReturn (Pirouette lp ap)) = (TurnMiddleReturn (Pirouette lp ap))
map_f f (TurnMiddleReturn (Fouette lp ap)) = (TurnMiddleReturn (Fouette lp ap))
map_f f (TurnMiddleReturn (Pique lp ap)) = (TurnMiddleReturn (Pique lp ap))
map_f f (TurnMiddleReturn (Chaine lp ap)) = (TurnMiddleReturn (Chaine lp ap))
map_f f (LeapPrepReturn (Chasse d)) = (LeapPrepReturn (Chasse d))
map_f f (LeapPrepReturn (StepStep d)) = (LeapPrepReturn (StepStep d))
map_f f (LeapMiddleReturn (Jete d lp ap)) = (LeapMiddleReturn (Jete d lp ap))
map_f f (LeapMiddleReturn (Switch d lp ap)) = (LeapMiddleReturn (Switch d lp ap))
map_f f (LeapMiddleReturn (CenterLeap d lp ap)) = (LeapMiddleReturn (CenterLeap d
map_f f (EndReturn (Land fp ap)) = (EndReturn (Land fp ap))
map_f f (BattementBegEndReturn (Step d)) = (BattementBegEndReturn (Step d))
map_f f (BattementMiddleReturn (Grande d lp ap)) = (BattementMiddleReturn (Grande
map_f f (BattementMiddleReturn (Petit d lp ap)) = (BattementMiddleReturn (Petit d lp
map_f f (RondDeJambeReturn (RondDeJambe d ap1 fp ap2)) = (RondDeJambeReturn (
map_f f (FillerReturn (Walk)) = (FillerReturn (Walk))
map_f f (FillerReturn (Hold)) = (FillerReturn (Hold))
map_f f (FillerReturn (Continue)) = (FillerReturn (Continue))

```

type *EightCount* $m = (Integer, [(TimedMovement\ m)])$

```
type RNRet = ([String])
```

```
type DescripRet = ([String])
```

```
type TimedMovement m = (MCountRet, (MovementType m))
```

```
data Movement = Direction  
    | FootPosition  
    | ArmPosition  
    | LegPosition  
    | TurnPrep  
    | TurnMiddle  
    | LeapPrep  
    | LeapMiddle  
    | End  
    | BattementBegEnd  
    | BattementMiddle  
    | RondDeJambe  
    | Filler
```

```
{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances -fallow-overlapping-instances #-}
```

```
module Dance2Types where  
import Dance2AST  
import PPrint  
import LangUtils  
import Dance2Environment  
import Dance2EvalMon  
import Monad  
import Control.Monad.Error
```

Trying to incorporate the Error monad into Type Checking.

First need to define the type of the type error representation. For now I'm just going to leave it as a String. Once I get this working I can eventually add other components here such as:

`data TypeError = Err location::MovementLang, reason::String` But for now I just want to concentrate on getting an error message to print out.

```
data TypeError = TErr{ reason :: String }  
                deriving (Eq, Show)
```

Now that the TypeError data structure is defined it must be made an instance of the Error class.

```
instance Error TypeError where  
    noMsg = TErr "Type Checking Error"  
    strMsg s = TErr s
```

```
instance MonadError (Either e) where  
    throwError = Err  
    catchError s handler = handler s  
    catchError a _ = a
```

I want to use Either for the monad type constructor (I think) so that failure is represented using Left TypeError and a success is represented using Right TypeError.

```
type TypeMonad = Either TypeError TyD
```

Must create a new data structure for the instances of Algebra to return. Now that I'm throwing errors using the Monad, I will have each instance of Algebra return a Value which is...

```
data TyValue = MType TyD  
             | Err String
```

```
instance Show TyValue where  
    show (MType mt) = "MType " ++ (show mt)  
    show (Err e) = "Err Message"
```

instance Subtype Ty_D TyValue where

↑ $t = MType\ t$
↓ $(MType\ t) = Just\ t$
↓ $(Err\ t) = Nothing$

instance Subtype String TyValue where

↑ $s = Err\ s$
↓ $(Err\ s) = Just\ s$
↓ $(MType\ s) = Nothing$

instance Algebra TurnPrep TypeMonad where

φ (Prepare Dance2AST.Right FourthF FourthA) = return \$ ↑ (TurnPrepTy (PrepareTy
φ (Prepare Dance2AST.Left FourthF FourthA) = return \$ ↑ (TurnPrepTy (PrepareTy
φ (Prepare d fp ap) = throwError (TErr ("This is not a typical Turn Preparation
φ (StepPrepare Dance2AST.Right SecondF SecondA FourthF FourthA) = return \$ ↑ (T
φ (StepPrepare Dance2AST.Left SecondF SecondA FourthF FourthA) = return \$ ↑ (Tu
φ (StepPrepare d fp1 ap1 fp2 ap2) = throwError (TErr ("This is not a typical Tu

instance Algebra TurnMiddle TypeMonad where

φ (Pirouette Coupe FirstA) = return \$ ↑ (TurnMiddleTy (PirouetteType Coupe))
φ (Pirouette Passe FirstA) = return \$ ↑ (TurnMiddleTy (PirouetteType Passe))
φ (Pirouette OpenPasse FirstA) = return \$ ↑ (TurnMiddleTy (PirouetteType OpenPasse))
φ (Pirouette Coupe SecondA) = return \$ ↑ (TurnMiddleTy (PirouetteType Coupe))
φ (Pirouette Passe HighV) = return \$ ↑ (TurnMiddleTy (PirouetteType Passe))
φ (Pirouette lp ap) = throwError (TErr ("This is not a typical Pirouette"))
φ (Fouette Straight SecondA) = return \$ ↑ (TurnMiddleTy (FouetteType Straight))
φ (Fouette Straight HighV) = return \$ ↑ (TurnMiddleTy (FouetteType Straight))
φ (Fouette lp ap) = throwError (TErr ("This is not a typical Fouette"))
φ (Chaine Straight FirstA) = return \$ ↑ (TurnMiddleTy (ChaineType Straight))
φ (Chaine lp ap) = throwError (TErr ("This is not a typical Chaine"))
φ (Pique OpenPasse FirstA) = return \$ ↑ (TurnMiddleTy (PiqueType OpenPasse))
φ (Pique lp ap) = throwError (TErr ("This is not a typical Pique"))

instance Algebra LeapPrep TypeMonad where

φ (Chasse Dance2AST.Right) = return \$ ↑ (LeapPrepTy (ChasseType Dance2AST.Right))
φ (Chasse Dance2AST.Left) = return \$ ↑ (LeapPrepTy (ChasseType Dance2AST.Left))
φ (Chasse Center) = throwError (TErr ("This is not a typical Chasse"))
φ (StepStep Dance2AST.Right) = return \$ ↑ (LeapPrepTy (StepStepType Dance2AST.R
φ (StepStep Dance2AST.Left) = return \$ ↑ (LeapPrepTy (StepStepType Dance2AST.L
φ (StepStep Center) = throwError (TErr ("This is not a typical Step Step"))

instance Algebra LeapMiddle TypeMonad where

ϕ (*Jete Dance2AST.Right Developpe HighV*) = return \$ ↑ (*LeapMiddleTy (JeteType Dance2AST.Right Developpe HighV)*)
 ϕ (*Jete Dance2AST.Left Developpe HighV*) = return \$ ↑ (*LeapMiddleTy (JeteType Dance2AST.Left Developpe HighV)*)
 ϕ (*Jete Dance2AST.Right Straight HighV*) = return \$ ↑ (*LeapMiddleTy (JeteType Dance2AST.Right Straight HighV)*)
 ϕ (*Jete Dance2AST.Left Straight HighV*) = return \$ ↑ (*LeapMiddleTy (JeteType Dance2AST.Left Straight HighV)*)
 ϕ (*Jete Dance2AST.Right Developpe FourthA*) = return \$ ↑ (*LeapMiddleTy (JeteType Dance2AST.Right Developpe FourthA)*)
 ϕ (*Jete Dance2AST.Left Developpe FourthA*) = return \$ ↑ (*LeapMiddleTy (JeteType Dance2AST.Left Developpe FourthA)*)
 ϕ (*Jete d lp ap*) = throwError (*TErr ("This is not a typical Jete")*)
 ϕ (*Switch Dance2AST.Right Straight FourthA*) = return \$ ↑ (*LeapMiddleTy (SwitchType Dance2AST.Right Straight FourthA)*)
 ϕ (*Switch Dance2AST.Left Straight FourthA*) = return \$ ↑ (*LeapMiddleTy (SwitchType Dance2AST.Left Straight FourthA)*)
 ϕ (*Switch d lp ap*) = throwError (*TErr ("This is not a typical Switch Leap")*)
 ϕ (*CenterLeap Dance2AST.Right Straight SecondA*) = return \$ ↑ (*LeapMiddleTy (CenterLeapType Dance2AST.Right Straight SecondA)*)
 ϕ (*CenterLeap Dance2AST.Left Straight SecondA*) = return \$ ↑ (*LeapMiddleTy (CenterLeapType Dance2AST.Left Straight SecondA)*)
 ϕ (*CenterLeap d lp ap*) = throwError (*TErr ("This is not a typical Center Leap")*)

instance Algebra End TypeMonad where

ϕ (*Land SecondF LowV*) = return \$ ↑ (*LandingTy (LandType SecondF)*)
 ϕ (*Land FourthF FirstA*) = return \$ ↑ (*LandingTy (LandType FourthF)*)
 ϕ (*Land FourthF LowV*) = return \$ ↑ (*LandingTy (LandType FourthF)*)
 ϕ (*Land fp ap*) = throwError (*TErr ("This is not a typical Landing")*)

instance Algebra BattementBegEnd TypeMonad where

ϕ (*Step Dance2AST.Right*) = return \$ ↑ (*KickPrepLandTy (StepType Dance2AST.Right)*)
 ϕ (*Step Dance2AST.Left*) = return \$ ↑ (*KickPrepLandTy (StepType Dance2AST.Left)*)
 ϕ (*Step Center*) = throwError (*TErr ("This is not a typical Kick Preparation")*)

instance Algebra BattementMiddle TypeMonad where

ϕ (*Grande Dance2AST.Right Developpe RDiagonal*) = return \$ ↑ (*KickMiddleTy (GrandeType Dance2AST.Right Developpe RDiagonal)*)
 ϕ (*Grande Dance2AST.Left Developpe LDiagonal*) = return \$ ↑ (*KickMiddleTy (GrandeType Dance2AST.Left Developpe LDiagonal)*)
 ϕ (*Grande Dance2AST.Right Straight RDiagonal*) = return \$ ↑ (*KickMiddleTy (GrandeType Dance2AST.Right Straight RDiagonal)*)
 ϕ (*Grande Dance2AST.Left Straight LDiagonal*) = return \$ ↑ (*KickMiddleTy (GrandeType Dance2AST.Left Straight LDiagonal)*)
 ϕ (*Grande d lp ap*) = throwError (*TErr ("This is not a typical Grande Battement")*)
 ϕ (*Petit Dance2AST.Right Developpe RDiagonal*) = return \$ ↑ (*KickMiddleTy (PetitType Dance2AST.Right Developpe RDiagonal)*)
 ϕ (*Petit Dance2AST.Left Developpe LDiagonal*) = return \$ ↑ (*KickMiddleTy (PetitType Dance2AST.Left Developpe LDiagonal)*)
 ϕ (*Petit Dance2AST.Right Straight RDiagonal*) = return \$ ↑ (*KickMiddleTy (PetitType Dance2AST.Right Straight RDiagonal)*)
 ϕ (*Petit Dance2AST.Left Straight RDiagonal*) = return \$ ↑ (*KickMiddleTy (PetitType Dance2AST.Left Straight RDiagonal)*)
 ϕ (*Petit d lp ap*) = throwError (*TErr ("This is not a typical Petit Battement")*)

instance Algebra RondDeJambe TypeMonad where

ϕ (RondDeJambe Dance2AST.Right FifthA SecondF LowV) = return \$ ↑ (RondeDeJambe Dance2AST.Right FifthA SecondF LowV)
 ϕ (RondDeJambe Dance2AST.Left FifthA SecondF LowV) = return \$ ↑ (RondeDeJambe Dance2AST.Left FifthA SecondF LowV)
 ϕ (RondDeJambe d ap1 fp ap2) = throwError (TErr ("This is not a typical RondDeJambe"))

instance Algebra Filler TypeMonad where

ϕ (Walk) = return \$ ↑ (FillerTy WalkType)
 ϕ (Hold) = return \$ ↑ (FillerTy HoldType)
 ϕ (Continue) = return \$ ↑ (FillerTy ContinueType)

This next section never quite worked correctly - I think it was all the nested cases that caused the problem. There's probably a better way to go about this. One that's not so messy!

```

instance Algebra Continuation TypeMonad where phi (Follows m1 m2) =
do m1' ¡- m1 ; m2' ¡- m2 ; case (prj m1') of (Just (TurnMiddleTy tm))
-¡ case (prj m2') of (Just (LandingTy (LandType SecondF))) -¡ return
inj(TurnEndTytm(LandTypeSecondF))(Just(LandingTyl))- > throwError(TErr("Thisisnotatypical
throwError(TErr("Therewasnosecondmovementgiven"))(Just(TurnPrepTytp))- >
case(prjm2')of(Just(TurnEndTytmte))- > return inj (TurnTy tp tm
te) Nothing -¡ throwError (TErr ("This is not a proper turn ending"))
(Just (LeapMiddleTy lm)) -¡ case (prj m2') of (Just (LandingTy (LandType
FourthF))) -¡ return inj(LeapEndTytm(LandTypeFourthF))(Just(LandingTyl))- >
throwError(TErr("Thisisnotatypicallandingforaleap"))Nothing- > throwError(TErr("Therewasnosecond
case(prjm2')of(Just(LeapEndTy(JeteTypeDance2AST.Left)le))- > throwError(TErr("Yourjeteleapmustbeinthesamedirectionasyourpreparation"))(Just(LeapEndTy(JeteTypeDance2AST.Left)le))- > return inj (LeapTy (ChasseType Dance2AST.Right) (JeteType Dance2AST.Right) le) (Just (LeapEndTy (SwitchType Dance2AST.Right) le)) -¡ return inj(LeapTy(ChasseTypeDance2AST.Right)le) (Just (LeapEndTy (CenterLeapType Dance2AST.Right) le)) -¡ throwError(TErr("Yourswitchleapmustbeinthesamedirectionasyourpreparation"))(Just(LeapEndTy(CenterLeapTypeDance2AST.Right)le)) -¡ throwError(TErr("Yourcenterleapmustbeinthesamedirectionasyourpreparation")) Nothing -¡ throwError (TErr ("There was no second movement given")) (Just (LeapPrepTy (ChasseType Dance2AST.Left))) -¡ case (prj m2') of (Just (LeapEndTy (JeteType Dance2AST.Right) le)) -¡ throwError (TErr ("Your jete leap must be in the same direction as the preparation")) (Just (LeapEndTy (JeteType Dance2AST.Left) le)) -¡ return inj(LeapTy(ChasseTypeDance2AST.Left)le) (Just (LeapEndTy (SwitchType Dance2AST.Left) le)) -¡ return inj (LeapTy (ChasseType Dance2AST.Left) (SwitchType Dance2AST.Left) le) (Just (LeapEndTy (CenterLeapType Dance2AST.Right) le)) -¡ throwError(TErr("Yourcenterleapmustbeinthesamedirectionasyourpreparation")) (Just (LeapEndTy (CenterLeapType Dance2AST.Left) le)) -¡ return

```

$inj(LeapTy(ChasseTypeDance2AST.Left)(CenterLeapTypeDance2AST.Left)le)Nothing - >$
 $throwError(TErr("Therewasnosecondmovementgiven"))(Just(LeapPrepTy(StepStepTypeDance2AST.Left)le) - > throwError(TErr("Your preparation is not correct")))$
 $return inj(LeapTy(StepStepTypeDance2AST.Right)(JeteTypeDance2AST.Right)le) (Just(LeapEndTy(SwitchTypeDance2AST.Right)le)) -_i return inj(LeapTy(StepStepTypeDance2AST.Right)le) (Just(LeapEndTy(SwitchTypeDance2AST.Right)le)) -_i$
 $throwError(TErr("Your switch leap must be in the same direction as your preparation"))(Just(LeapPrepTy(StepStepTypeDance2AST.Right)le) - > throwError(TErr("Your preparation is not correct")))$
 $throwError(TErr("This is not a typical center leap preparation"))(Just(LeapEndTy(CenterLeapTypeDance2AST.Right)le)) - > throwError(TErr("This is not a typical center leap preparation, and the direction of your leap must be correct"))$
 $throwError(TErr("Therewasnosecondmovementgiven"))(Just(LeapPrepTy(StepStepTypeDance2AST.Right)le) - > throwError(TErr("Your preparation is not correct")))$
 $return inj(LeapTy(StepStepTypeDance2AST.Left)(JeteTypeDance2AST.Left)le) (Just(LeapEndTy(SwitchTypeDance2AST.Left)le)) -_i throwError(TErr("Your switch leap must be in the same direction as the preparation"))$
 $(Just(LeapEndTy(SwitchTypeDance2AST.Left)le)) -_i return inj(LeapTy(StepStepTypeDance2AST.Left)(SwitchTypeDance2AST.Left)le) (Just(LeapEndTy(SwitchTypeDance2AST.Left)le)) -_i$
 $throwError(TErr("This is not a typical center leap preparation"))(Just(LeapEndTy(CenterLeapTypeDance2AST.Left)le)) - > throwError(TErr("This is not a typical center leap preparation, and the direction of your leap must be correct"))$
 $throwError(TErr("Therewasnosecondmovementgiven"))(Just(KickMiddleTy(GrandeTypeDance2AST.Left)le) - > throwError(TErr("Your preparation is not correct")))$
 $return inj(KickEndTy(GrandeTypeDance2AST.Right)(StepTypeDance2AST.Right)le) (Just(KickPrepLandTy(StepTypeDance2AST.Right)le)) - >$
 $return inj(KickEndTy(GrandeTypeDance2AST.Right)(StepTypeDance2AST.Right)le) (Just(KickPrepLandTy(StepTypeDance2AST.Right)le)) -_i throwError(TErr("You must land with the same leg you kicked with"))$
 $Nothing -_i throwError(TErr("There was no second movement given"))(Just(KickMiddleTy(GrandeTypeDance2AST.Left)le)) -_i case (prj m2') of (Just(KickPrepLandTy(StepTypeDance2AST.Left)le)) -_i$
 $return inj(KickEndTy(GrandeTypeDance2AST.Left)le) (StepTypeDance2AST.Left)le) (Just(KickPrepLandTy(StepTypeDance2AST.Left)le)) -_i return inj(KickEndTy(GrandeTypeDance2AST.Left)le) (StepTypeDance2AST.Left)le) (Just(KickPrepLandTy(StepTypeDance2AST.Left)le)) -_i$
 $throwError(TErr("You must land with the same leg you kicked with"))$
 $Nothing - > throwError(TErr("Therewasnosecondmovementgiven"))(Just(KickMiddleTy(PetitTypeDance2AST.Left)le) - > throwError(TErr("Your preparation is not correct")))$
 $return inj(KickEndTy(PetitTypeDance2AST.Right)(StepTypeDance2AST.Right)le) (Just(KickPrepLandTy(StepTypeDance2AST.Right)le)) - >$
 $return inj(KickEndTy(PetitTypeDance2AST.Right)(StepTypeDance2AST.Right)le) (Just(KickPrepLandTy(StepTypeDance2AST.Right)le)) -_i throwError(TErr("You must land with the same leg you kicked with"))$
 $Nothing -_i throwError(TErr("There was no second movement given"))(Just(KickMiddleTy(PetitTypeDance2AST.Left)le)) -_i case (prj m2') of (Just(KickPrepLandTy(StepTypeDance2AST.Left)le)) -_i$
 $return inj(KickEndTy(PetitTypeDance2AST.Left)le) (StepTypeDance2AST.Left)le) (Just(KickPrepLandTy(StepTypeDance2AST.Left)le)) -_i return inj(KickEndTy(PetitTypeDance2AST.Left)le) (StepTypeDance2AST.Left)le) (Just(KickPrepLandTy(StepTypeDance2AST.Left)le)) -_i$
 $throwError(TErr("You must land with the same leg you kicked with"))$
 $Nothing - > throwError(TErr("Therewasnosecondmovementgiven"))(Just(KickPrepLandTy(StepTypeDance2AST.Right)le) - > throwError(TErr("Your preparation is not correct")))$
 $return inj(KickTy(StepTypeDance2AST.Left)(GrandeTypeDance2AST.Right)le) (StepTypeDance2AST.Right)le) (Just(KickEndTy(PetitTypeDance2AST.Right)le) (StepTypeDance2AST.Right)le)) -_i return inj(KickTy(StepTypeDance2AST.Left)(PetitTypeDance2AST.Right)le) (StepTypeDance2AST.Right)le) (Just(KickEndTy(PetitTypeDance2AST.Right)le) (StepTypeDance2AST.Right)le)) -_i$

```

throwError (TErr ("Therewasnosecondmovementgiven")) (Just (KickPrepLandTy (StepTypeDance2AST.Right)
case (prjm2') of (Just (KickEndTy (GrandeTypeDance2AST.Left) (StepTypeDance2AST.Left))
return inj (KickTy (StepTypeDance2AST.Right) (GrandeTypeDance2AST.Left)
(StepTypeDance2AST.Left)) (Just (KickEndTy (PetitTypeDance2AST.Left)
(StepTypeDance2AST.Left))) -; return inj (KickTy (StepTypeDance2AST.Right) (PetitTypeDance2AST.Left)
throwError (TErr ("Therewasnosecondmovementgiven")) (Just (KickPrepLandTy kpl)) - >
case (prjm2') of (Just (KickEndTy klt)) - > throwError (TErr ("Youmustkickwiththeoppositeleg"))
throwError (TErr ("Therewasnosecondmovementgiven")) Nothing - > throwError (TErr ("Therewasnosecondmovementgiven"))

```

-Combining Full Movements-

instance Algebra VarTerm TypeMonad where

```

phi (Var s) = throwError (TErr ("Don't have this defined yet"))

```

instance Algebra FcnTerm TypeMonad where

```

phi (Lambda s e) = throwError (TErr ("Don't have this defined yet"))
phi (App a1 a2) = throwError (TErr ("Don't have this defined yet"))

```

instance Algebra MovementType TypeMonad where

```

phi (DirectionReturn d) = throwError (TErr ("Don't have this defined yet"))
phi (FootPosReturn fp) = throwError (TErr ("Don't have this defined yet"))
phi (ArmPosReturn ap) = throwError (TErr ("Don't have this defined yet"))
phi (LegPosReturn lp) = throwError (TErr ("Don't have this defined yet"))
phi (TurnPrepReturn (Prepare d lp ap)) = throwError (TErr ("Don't have this defined yet"))
phi (TurnPrepReturn (StepPrepare d lp1 ap1 lp2 ap2)) = throwError (TErr ("Don't have this defined yet"))
phi (TurnMiddleReturn (Pirouette lp ap)) = throwError (TErr ("Don't have this defined yet"))
phi (TurnMiddleReturn (Fouette lp ap)) = throwError (TErr ("Don't have this defined yet"))
phi (TurnMiddleReturn (Pique lp ap)) = throwError (TErr ("Don't have this defined yet"))
phi (TurnMiddleReturn (Chaine lp ap)) = throwError (TErr ("Don't have this defined yet"))
phi (LeapPrepReturn (Chasse d)) = throwError (TErr ("Don't have this defined yet"))
phi (LeapPrepReturn (StepStep d)) = throwError (TErr ("Don't have this defined yet"))
phi (LeapMiddleReturn (Jete d lp ap)) = throwError (TErr ("Don't have this defined yet"))
phi (LeapMiddleReturn (Switch d lp ap)) = throwError (TErr ("Don't have this defined yet"))
phi (LeapMiddleReturn (CenterLeap d lp ap)) = throwError (TErr ("Don't have this defined yet"))
phi (EndReturn (Land fp ap)) = throwError (TErr ("Don't have this defined yet"))
phi (BattementBegEndReturn (Step d)) = throwError (TErr ("Don't have this defined yet"))
phi (BattementMiddleReturn (Grande d lp ap)) = throwError (TErr ("Don't have this defined yet"))
phi (BattementMiddleReturn (Petit d lp ap)) = throwError (TErr ("Don't have this defined yet"))
phi (RondDeJambeReturn (RondDeJambe d ap1 fp ap2)) = throwError (TErr ("Don't have this defined yet"))
phi (FillerReturn (Walk)) = throwError (TErr ("Don't have this defined yet"))

```

```

     $\phi$  (FillerReturn (Hold)) = throwError (TErr ("Don't have this defined yet"))
     $\phi$  (FillerReturn (Continue)) = throwError (TErr ("Don't have this defined yet"))

typeof $\mathcal{D}$  :: MovementLang → TypeMonad
typeof $\mathcal{D}$  = cata

-- thisrocks s = (typeof (eval (toMovementLang (Var s))))

-- thisrocks2 m1 m2 = typeof (eval (toMovementLang (Follows (thisrocks m1 thisrocks m2))))

onlyEval s = (eval $\mathcal{D}$  (toMovementLang (Var s)))

```

3 Parsing and Concrete Syntax

This section contains some experimental parser definitions. Nothing is set here - the objective is simply to demonstrate what might be parsed. Some of the functions for parsing some basic keywords should be reusable.

```

module Dance2Parser where

import System
import Text.ParserCombinators.Parsec
import Text.ParserCombinators.Parsec.Expr
import Text.ParserCombinators.Parsec.Token
import Text.ParserCombinators.Parsec.Language
import Ratio
import Dance2PPrint
import Dance2AST

```

3.1 Lexer Definition

A *TokenParser*, or lexer, is really a data structure that provides a way to write mini-parsers for each of the specified keywords and operators. When a new keyword or operator is added to the concrete syntax, add it to the appropriate list.

```

lexer :: TokenParser ()
lexer = makeTokenParser
      (javaStyle
       { reservedNames = ["Routine", "Name", "Style", "Number", "of", "Dancers", "Jaz",
                          "Novelty", "Pom", "Prop", "Beginning", "Position", "Ending",
                          , reservedOpNames = ["+", "*", "||", ":", "-", ", ", "&", "(, )", ";"]
       })

```

3.2 Token Parsers

Functions for recognizing tokens are provided in the *Token* package and then instantiated here. Using the *lexer* as a parameter carries the respective function to make including the *lexer* parameter in every call. The Parsec documentation includes an example with a qualified import that makes it unnecessary to qualify names with "P". Look at that later. Another option would be to put the parser in a different package than the abstract syntax definition. That would clean things up nicely.

```

whiteSpaceP = whiteSpace lexer
parensP = parens lexer
reservedP = reserved lexer
operatorP = reservedOp lexer
plusP = reservedP "+"
timesP = reservedP "*"
whileP = reservedP "||"
dashP = operatorP "-"
commaP = comma lexer
andP = reservedP "&"
equalP = operatorP "="
colonP = colon lexer
routineP = reservedP "Routine"
nameP = reservedP "Name"
styleP = reservedP "Style"
numberP = reservedP "Number"
ofP = reservedP "of"
dancersP = reservedP "Dancers"
beginningP = reservedP "Beginning"
positionP = reservedP "Position"
endingP = reservedP "Ending"

```

```

commaSep1P = commaSep1 lexer
stringP = idP
jazzP = reservedP "Jazz"
hipHopP = reservedP "Hip-Hop"
lyricalP = reservedP "Lyrical"
noveltyP = reservedP "Novelty"
pomP = reservedP "Pom"
propP = reservedP "Prop"
idP = identifier lexer
moveP = reservedP "movement"
intP = natural lexer
semiP = semi lexer

```

3.3 Basic Value Parsers

```

programP :: Parser (Routine MovementLang)
programP = do { whiteSpaceP
               ; rn ← routineNameP
               ; rs ← routineStyleP
               ; nod ← numOfDancersP
               ; bp ← begPosP
               ; l ← many lineP
               ; ep ← endPosP
               ; return (Routine rn rs nod bp l ep) }

```

```

routineNameP :: Parser RNRet
routineNameP = do { try routineP
                  ; nameP
                  ; colonP
                  ; sl ← wordList
                  ; semiP
                  ; return sl }

```

```

styleInnerP :: Parser StyleRet
styleInnerP = do { s ← jazzP
                    ; return Jazz }
< | >
do { s ← hipHopP
      ; return HipHop }
< | >
do { s ← lyricalP
      ; return Lyrical }
< | >
do { s ← noveltyP
      ; return Novelty }
< | >
do { s ← pomP
      ; return Pom }
< | >
do { s ← propP
      ; return Prop }

```

```

routineStyleP :: Parser StyleRet
routineStyleP = do { try routineP
                    ; styleP
                    ; colonP
                    ; s ← styleInnerP
                    ; return s }

```

```

numOfDancersP :: Parser Integer
numOfDancersP = do { numberP
                    ; ofP
                    ; dancersP
                    ; colonP
                    ; i1 ← intP
                    ; return i1 }

```

```
begPosP :: Parser DescripRet
begPosP = do { beginningP
                ; positionP
                ; colonP
                ; d ← descriptionP
                ; semiP
                ; return d }
```

```
endPosP :: Parser DescripRet
endPosP = do { endingP
                ; positionP
                ; colonP
                ; d ← descriptionP
                ; semiP
                ; return d }
```

```
descriptionP :: Parser DescripRet
descriptionP = wordList
```

```
lCountP :: Parser Integer
lCountP = do { i ← intP
                ; colonP
                ; return i }
```

```
mCountP :: Parser MCountRet
mCountP = do { mc ← try (parensP mCountInnerDash)
                ; return (mc) }
< | >
do { mc ← try (parensP mCountInnerComma)
      ; return (mc) }
< | >
```

```

do { mc ← try (parensP mCountInnerIntAnd)
      ; return (mc) }
< | >
do { mc ← try (parensP mCountInnerAnd)
      ; return (mc) }
< | >
do { mc ← try (parensP mCountInnerInt)
      ; return (mc) }

```

```

mCountInnerDash :: Parser MCountRet
mCountInnerDash = do { i1 ← intP
                       ; dashP
                       ; i2 ← intP
                       ; return (MCDash i1 i2) }

```

```

mCountInnerComma :: Parser MCountRet
mCountInnerComma = do { i1 ← intP
                       ; commaP
                       ; i2 ← intP
                       ; return (MCComma i1 i2) }

```

```

mCountInnerIntAnd :: Parser MCountRet
mCountInnerIntAnd = do { i1 ← intP
                       ; andP
                       ; return (MCIntAnd i1) }

```

```

mCountInnerAnd :: Parser MCountRet
mCountInnerAnd = do { andP
                       ; return MCAnd }

```

```
mCountInnerInt :: Parser MCountRet
mCountInnerInt = do { i1 ← intP
                      ; return (MInt i1) }
```

```
lineMoreP :: Parser (TimedMovement MovementLang)
lineMoreP = do { mc ← mCountP
                  ; m ← movementP
                  ; optional semiP
                  ; return (mc, m) }
```

```
lineP :: Parser (EightCount MovementLang)
lineP = do { lc ← lCountP
              ; lm ← many lineMoreP
              ; return (lc, lm) }
```

```
directionP :: Parser Direction
directionP = do { reservedP "right"
                  ; return Dance2AST.Right }
< | >
do { reservedP "left"
      ; return Dance2AST.Left }
< | >
do { reservedP "center"
      ; return Center }
```

```
footPositionP :: Parser FootPosition
footPositionP = do { reservedP "first"
                    ; return FirstF }
< | >
do { reservedP "second"
      ; return SecondF }
```

```

< | >
do { reservedP "third"
    ; return ThirdF }
< | >
do { reservedP "fourth"
    ; return FourthF }
< | >
do { reservedP "fifth"
    ; return FifthF }
< | >
do { reservedP "prevF"
    ; return PrevF }
< | >
do { reservedP "nextF"
    ; return NextF }

```

```

armPositionP :: Parser ArmPosition
armPositionP = do { reservedP "first"
    ; return FirstA }
< | >
do { reservedP "second"
    ; return SecondA }
< | >
do { reservedP "third"
    ; return ThirdA }
< | >
do { reservedP "fourth"
    ; return FourthA }
< | >
do { reservedP "fifth"
    ; return FifthA }
< | >
do { reservedP "highV"
    ; return HighV }
< | >
do { reservedP "LowV"
    ; return LowV }

```

```

< | >
do { reservedP "tee"
    ; return Tee }
< | >
do { reservedP "rdiagonal"
    ; return RDiagonal }
< | >
do { reservedP "ldiagonal"
    ; return LDiagonal }
< | >
do { reservedP "prevA"
    ; return PrevA }
< | >
do { reservedP "nextA"
    ; return NextA }

```

```

legPositionP :: Parser LegPosition
legPositionP = do { reservedP "attitude"
    ; return Attitude }
< | >
do { reservedP "coupe"
    ; return Coupe }
< | >
do { reservedP "passe"
    ; return Passe }
< | >
do { reservedP "open-passe"
    ; return OpenPasse }
< | >
do { reservedP "develope"
    ; return Develope }
< | >
do { reservedP "straight"
    ; return Straight }

```

```

turnMiddleP :: Parser (TurnMiddle m)
turnMiddleP = do { reservedP "Pirouette"
                  ; reservedP "leg"
                  ; equalP
                  ; lp ← legPositionP
                  ; reservedP "arms"
                  ; equalP
                  ; ap ← armPositionP
                  ; return (Pirouette lp ap) }
< | >
do { reservedP "Fouette"
    ; reservedP "leg"
    ; equalP
    ; lp ← legPositionP
    ; reservedP "arms"
    ; equalP
    ; ap ← armPositionP
    ; return (Fouette lp ap) }
< | >
do { reservedP "Chaine"
    ; reservedP "leg"
    ; equalP
    ; lp ← legPositionP
    ; reservedP "arms"
    ; equalP
    ; ap ← armPositionP
    ; return (Chaine lp ap) }
< | >
do { reservedP "Pique"
    ; reservedP "leg"
    ; equalP
    ; lp ← legPositionP
    ; reservedP "arms"
    ; equalP
    ; ap ← armPositionP
    ; return (Pique lp ap) }

```

```

turnPrepP :: Parser (TurnPrep m)

```

```

turnPrepP = do { reservedP "Prepare"
                ; d ← directionP
                ; reservedP "feet"
                ; equalP
                ; fp ← footPositionP
                ; reservedP "arms"
                ; equalP
                ; ap ← armPositionP
                ; return (Prepare d fp ap) }
< | >
do { reservedP "Step-Prepare"
    ; d ← directionP
    ; reservedP "feet"
    ; equalP
    ; fp1 ← footPositionP
    ; reservedP "arms"
    ; equalP
    ; ap1 ← armPositionP
    ; reservedP "feet"
    ; equalP
    ; fp2 ← footPositionP
    ; reservedP "arms"
    ; equalP
    ; ap2 ← armPositionP
    ; return (StepPrepare d fp1 ap1 fp2 ap2) }

```

```

leapPrepP :: Parser (LeapPrep m)
leapPrepP = do { optional mCountP
                ; reservedP "Chasse"
                ; d ← directionP
                ; return (Chasse d) }
< | >
do { optional mCountP
    ; reservedP "Step-Step"
    ; d ← directionP
    ; return (StepStep d) }

```

```

endP :: Parser (End m)
endP = do { optional mCountP
           ; reservedP "Land"
           ; reservedP "feet"
           ; equalP
           ; fp ← footPositionP
           ; reservedP "arms"
           ; equalP
           ; ap ← armPositionP
           ; return (Land fp ap) }

```

```

leapMiddleP :: Parser (LeapMiddle m)
leapMiddleP = do { reservedP "Jete"
                  ; d ← directionP
                  ; reservedP "legs"
                  ; equalP
                  ; lp ← legPositionP
                  ; reservedP "arms"
                  ; equalP
                  ; ap ← armPositionP
                  ; return (Jete d lp ap) }
< | >
do { reservedP "Switch Leap"
    ; d ← directionP
    ; reservedP "legs"
    ; equalP
    ; lp ← legPositionP
    ; reservedP "arms"
    ; equalP
    ; ap ← armPositionP
    ; return (Switch d lp ap) }
< | >
do { reservedP "Center Leap"
    ; d ← directionP
    ; reservedP "legs"
    ; equalP
    ; lp ← legPositionP

```

```
; reservedP "arms"  
; equalP  
; ap ← armPositionP  
; return (CenterLeap d lp ap)}
```

```
battementBegEndP :: Parser (BattementBegEnd m)  
battementBegEndP = do { optional mCountP  
; reservedP "Step"  
; d ← directionP  
; return (Step d)}
```

```
battementMiddleP :: Parser (BattementMiddle m)  
battementMiddleP = do { reservedP "Grande Battement"  
; d ← directionP  
; reservedP "leg"  
; equalP  
; lp ← legPositionP  
; reservedP "arms"  
; equalP  
; ap ← armPositionP  
; return (Grande d lp ap)}  
< | >  
do { reservedP "Petit Battement"  
; d ← directionP  
; reservedP "leg"  
; equalP  
; lp ← legPositionP  
; reservedP "arms"  
; equalP  
; ap ← armPositionP  
; return (Petit d lp ap)}
```

```

rondDeJambeP :: Parser (RondDeJambe m)
rondDeJambeP = do { reservedP "Rond De Jambe"
                    ; d ← directionP
                    ; reservedP "arms"
                    ; equalP
                    ; ap1 ← armPositionP
                    ; optional mCountP
                    ; reservedP "Land"
                    ; reservedP "feet"
                    ; equalP
                    ; fp ← footPositionP
                    ; reservedP "arms"
                    ; equalP
                    ; ap2 ← armPositionP
                    ; return (RondDeJambe d ap1 fp ap2)}

```

```

fillerP :: Parser (Filler MovementLang)
fillerP = do { reservedP "Hold"
                ; return (Hold)}
< | >
do { reservedP "Walk"
      ; return (Walk)}
< | >
do { reservedP "Continue"
      ; return (Continue)}

```

```

continuationP :: Parser (Continuation (MovementType MovementLang))
continuationP = do { optional mCountP
                    ; m1 ← movementP
                    ; reservedP "followed by"
                    ; optional mCountP
                    ; m2 ← movementP
                    ; return (Follows m1 m2)}
< | >
do { optional mCountP

```

```

; m1 ← movementP
; reservedP "while"
; optional mCountP
; m2 ← movementP
; return (While m1 m2)

```

movementP :: Parser (MovementType MovementLang)

```

movementP = do { m ← directionP
                ; return (DirectionReturn m) }
< | >
do { m ← footPositionP
    ; return (FootPosReturn m) }
< | >
do { m ← armPositionP
    ; return (ArmPosReturn m) }
< | >
do { m ← legPositionP
    ; return (LegPosReturn m) }
< | >
do { m ← turnPrepP
    ; return (TurnPrepReturn m) }
< | >
do { m ← turnMiddleP
    ; return (TurnMiddleReturn m) }
< | >
do { m ← leapPrepP
    ; return (LeapPrepReturn m) }
< | >
do { m ← leapMiddleP
    ; return (LeapMiddleReturn m) }
< | >
do { m ← endP
    ; return (EndReturn m) }
< | >
do { m ← battementBegEndP
    ; return (BattementBegEndReturn m) }
< | >

```

```

do { m ← battementMiddleP
      ; return (BattementMiddleReturn m) }
< | >
do { m ← rondDeJambeP
      ; return (RondDeJambeReturn m) }
< | >
do { m ← fillerP
      ; return (FillerReturn m) }

```

```

wordList :: Parser [String]
wordList = commaSep lexer (many1 (letter < | > digit < | > char ' '))

```

3.4 Main Parser

```

main = do { s : _ ← getArgs
             ; parseDance s
           }

parseDance s = do { r ← parseFromFile programP s
                    ; case (r) of
                      Prelude.Left err → print err
                      Prelude.Right mov → print (getMoves mov)
                    }

```

```

getMoves (Routine _ _ _ _ l _) = getMove l

```

```

getMove [] = []
getMove ((y, ys) : xs) = [y | (x, y) ← ys] ++ getMove xs

```

```

{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances -fallow-overlapping-instances #-}

```

```

module Dance2PPrint where
import PPrint
import Dance2AST
import LangUtils
import Dance2EvalMon

```

– Can combine this into `routineStylePP`
and get rid of `styleInnerPP`

```

styleInnerPP :: StyleRet → Doc
styleInnerPP Jazz = text "Jazz"
styleInnerPP HipHop = text "Hip Hop"
styleInnerPP Lyrical = text "Lyrical"
styleInnerPP Novelty = text "Novelty"
styleInnerPP Pom = text "Pom"
styleInnerPP Prop = text "Prop"

```

```

routineStylePP :: StyleRet → Doc
routineStylePP (s) = linebreak < + > text "Routine Style:" < + > (styleInnerPP s)

```

```

beatStructurePP :: BeatStructure → Doc
beatStructurePP Beat = text "duration"

```

```

numOfDancersPP :: Integer → Doc
numOfDancersPP (n) = linebreak < + > text "Number of Dancers:" < + > integer n < + >

```

```

mCountPP :: MCountRet → Doc
mCountPP (MCDash c1 c2) = indent 6 ((parens (integer c1 < + > text "-" < + > integer c2)))
mCountPP (MComma c1 c2) = indent 6 ((parens (integer c1 < + > text "," < + > integer c2)))
mCountPP (MCIntAnd c1) = indent 6 ((parens (integer c1 < + > text "&")))
mCountPP MCAnd = indent 6 (((parens (text "&"))))
mCountPP (MCInt c1) = indent 6 ((parens (integer c1)))

```

```

-- This actually needs to take in an RNRet...but I couldn't get it to compile
routineNamePP :: RNRet → Doc
routineNamePP (rn) = linebreak < + > text "Routine Name:" < + > (foldl (< + >) empty

```

```

-- This should take in a DescripRet
begPosPP :: DescripRet → Doc
begPosPP (bp) = linebreak < + > text "Beginning Position:" < + > foldl (< + >) empty

```

```

endPosPP :: DescripRet → Doc
endPosPP (ep) = linebreak < + > linebreak < + > text "Ending Position:" < + > foldl (<

```

```

lineMorePP :: (TimedMovement MovementLang) → Doc
lineMorePP (mc, (DirectionReturn ms)) = (mCountPP mc) < + > (directionPP ms)
lineMorePP (mc, (FootPosReturn ms)) = (mCountPP mc) < + > (footPosPP ms)
lineMorePP (mc, (ArmPosReturn ms)) = (mCountPP mc) < + > (armPosPP ms)
lineMorePP (mc, (LegPosReturn ms)) = (mCountPP mc) < + > (legPosPP ms)

lineMorePP (mc, (TurnPrepReturn ms)) = (mCountPP mc) < + > (pprint (toMovementLang ms))
lineMorePP (mc, (TurnMiddleReturn ms)) = (mCountPP mc) < + > (pprint (toMovementLang ms))
lineMorePP (mc, (LeapPrepReturn ms)) = (mCountPP mc) < + > (pprint (toMovementLang ms))
lineMorePP (mc, (LeapMiddleReturn ms)) = (mCountPP mc) < + > (pprint (toMovementLang ms))
lineMorePP (mc, (EndReturn ms)) = (mCountPP mc) < + > (pprint (toMovementLang ms))
lineMorePP (mc, (BattementBegEndReturn ms)) = (mCountPP mc) < + > (pprint (toMovementLang ms))
lineMorePP (mc, (BattementMiddleReturn ms)) = (mCountPP mc) < + > (pprint (toMovementLang ms))
lineMorePP (mc, (RondDeJambeReturn ms)) = (mCountPP mc) < + > (pprint (toMovementLang ms))
lineMorePP (mc, (FillerReturn ms)) = (mCountPP mc) < + > (pprint (toMovementLang ms))

```

```

lCountPP :: Integer → Doc
lCountPP (l) = integer l

```

linePP :: (*EightCount MovementLang*) → *Doc*
linePP (*lc, lm*) = *linebreak* <> (*lCountPP lc*) <> *text* ":" < + > *foldl* (< + >) *empty* (*map*

programPP :: (*Routine MovementLang*) → *Doc*
programPP (*Routine rn rs nd bp lp ep*) = *foldl* (< + >) *empty* (*map routineNamePP [rn]*) <

—————Direction Pretty Printer—————

directionPP :: *Direction* → *Doc*
directionPP *Dance2AST.Right* = *text* "Right"
directionPP *Dance2AST.Left* = *text* "Left"
directionPP *Center* = *text* "Center"

—————Foot Position Pretty Printer—————

footPosPP :: *FootPosition* → *Doc*
footPosPP *FirstF* = *text* "First"
footPosPP *SecondF* = *text* "Second"
footPosPP *ThirdF* = *text* "Third"
footPosPP *FourthF* = *text* "Fourth"
footPosPP *FifthF* = *text* "Fifth"

—————Arm Position Pretty Printer—————

armPosPP :: *ArmPosition* → *Doc*
armPosPP *FirstA* = *text* "First"
armPosPP *SecondA* = *text* "Second"
armPosPP *ThirdA* = *text* "Third"
armPosPP *FourthA* = *text* "Fourth"
armPosPP *FifthA* = *text* "Fifth"
armPosPP *HighV* = *text* "High V"
armPosPP *LowV* = *text* "Low V"
armPosPP *Tee* = *text* "Tee"

—————Leg Position Pretty Printer—————

```
legPosPP :: LegPosition → Doc
legPosPP Attitude = text "Attitude"
legPosPP Coupe = text "Coupe"
legPosPP Passe = text "Passe"
legPosPP OpenPasse = text "Open Passe"
legPosPP Develope = text "Develope"
legPosPP Straight = text "Straight"
```

—————Class PrettyPrint—————

—————Turn Prep instance—————

```
instance Algebra TurnPrep Doc where
  φ (Prepare d fp ap) = text "Prepare" < + > (directionPP d) < + > text "with feet in"
  φ (StepPrepare d fp1 ap1 fp2 ap2) = text "Step" < + > (directionPP d) < + > text "to"
```

—————Turn Middle instance—————

```
instance Algebra TurnMiddle Doc where
  φ (Pirouette lp ap) = text "Pirouette with leg in" < + > (legPosPP lp) < + > text "and arm"
  φ (Fouette lp ap) = text "Fouette with leg in" < + > (legPosPP lp) < + > text "and arm"
  φ (Chaine lp ap) = text "Chaine with leg in" < + > (legPosPP lp) < + > text "and arm"
  φ (Pique lp ap) = text "Pique with leg in" < + > (legPosPP lp) < + > text "and arm"
```

—————Leap Prep instance—————

```
instance Algebra LeapPrep Doc where
  φ (Chasse d) = text "Chasse" < + > (directionPP d)
  φ (StepStep d) = text "Step" < + > (directionPP d)
```

—————Leap Middle instance—————

```
instance Algebra LeapMiddle Doc where
  φ (Jete d lp ap) = (directionPP d) < + > text "Jete - Legs" < + > (legPosPP lp) < + > text "and arm"
  φ (Switch d lp ap) = (directionPP d) < + > text "Switch Leap - Legs" < + > (legPosPP lp) < + > text "and arm"
  φ (CenterLeap d lp ap) = (directionPP d) < + > text "CenterLeap - Legs" < + > (legPosPP lp) < + > text "and arm"
```

-----End Position instance-----

instance Algebra End Doc where

$\phi (Land\ fp\ ap) = text\ "Land\ with\ feet\ in" < + > (footPosPP\ fp) < + > text\ "and\ ar"$

-----Battement Prep/End instance-----

instance Algebra BattementBegEnd Doc where

$\phi (Step\ d) = text\ "Step" < + > (directionPP\ d)$

-----Battement Middle instance-----

instance Algebra BattementMiddle Doc where

$\phi (Grande\ d\ lp\ ap) = text\ "Kick" < + > (directionPP\ d) < + > text\ "leg" < + > (legP$

$\phi (Petit\ d\ lp\ ap) = text\ "Kick" < + > (directionPP\ d) < + > text\ "leg" < + > (legPosi$

-----Rond De Jambe instance-----

instance Algebra RondDeJambe Doc where

$\phi (RondDeJambe\ d\ ap1\ fp\ ap2) = text\ "Chaine" < + > (directionPP\ d) < + > text\ "Ro$

-----Filler instance-----

instance Algebra Filler Doc where

$\phi\ Walk = text\ "Walk"$

$\phi\ Hold = text\ "Hold"$

$\phi\ Continue = text\ "Continue"$

-----Continuation instance-----

instance Algebra Continuation Doc where

$\phi (Follows\ m_1\ m_2) = m_1 < + > text\ "Followed\ by" < + > m_2$

$\phi (While\ m_1\ m_2) = m_1 < + > text\ "While" < + > m_2$

instance Algebra VarTerm Doc where

$\phi (Var\ s) = text\ "Don't\ care\ about\ this"$

```

instance Algebra FcnTerm Doc where
   $\phi$  (Lambda s a) = text "Don't need to print this out"
   $\phi$  (App a1 a2) = text "Don't need to print this out either"

```

```

instance Algebra MovementType Doc where
   $\phi$  (DirectionReturn d) = text "Don't need to print this out"
   $\phi$  (FootPosReturn fp) = text "Don't need to print this out"
   $\phi$  (ArmPosReturn ap) = text "Don't need to print this out"
   $\phi$  (LegPosReturn lp) = text "Don't need to print this out"
   $\phi$  (TurnPrepReturn (Prepare d lp ap)) = text "Don't need to print this out"
   $\phi$  (TurnPrepReturn (StepPrepare d lp1 ap1 lp2 ap2)) = text "Don't need to print this out"
   $\phi$  (TurnMiddleReturn (Pirouette lp ap)) = text "Don't need to print this out"
   $\phi$  (TurnMiddleReturn (Fouette lp ap)) = text "Don't need to print this out"
   $\phi$  (TurnMiddleReturn (Pique lp ap)) = text "Don't need to print this out"
   $\phi$  (TurnMiddleReturn (Chaine lp ap)) = text "Don't need to print this out"
   $\phi$  (LeapPrepReturn (Chasse d)) = text "Don't need to print this out"
   $\phi$  (LeapPrepReturn (StepStep d)) = text "Don't need to print this out"
   $\phi$  (LeapMiddleReturn (Jete d lp ap)) = text "Don't need to print this out"
   $\phi$  (LeapMiddleReturn (Switch d lp ap)) = text "Don't need to print this out"
   $\phi$  (LeapMiddleReturn (CenterLeap d lp ap)) = text "Don't need to print this out"
   $\phi$  (EndReturn (Land fp ap)) = text "Don't need to print this out"
   $\phi$  (BattementBegEndReturn (Step d)) = text "Don't need to print this out"
   $\phi$  (BattementMiddleReturn (Grande d lp ap)) = text "Don't need to print this out"
   $\phi$  (BattementMiddleReturn (Petit d lp ap)) = text "Don't need to print this out"
   $\phi$  (RondDeJambeReturn (RondDeJambe d ap1 fp ap2)) = text "Don't need to print this out"
   $\phi$  (FillerReturn (Walk)) = text "Don't need to print this out"
   $\phi$  (FillerReturn (Hold)) = text "Don't need to print this out"
   $\phi$  (FillerReturn (Continue)) = text "Don't need to print this out"

```

-Cata-

```

pprint :: MovementLang → Doc
pprint = cata

```

```

{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances -fallow-overlapping-instances #-}

```

4 Monadic Evaluator

We define a monadic evaluator using a *Reader* monad to manage the environment during execution. The *Reader* is initialized with the *movements*

injected into the value space in a *Reader* monad. This is achieved using the *return* and \uparrow functions respectively.

instance Algebra MovementType MovEnvErr where

$$\begin{aligned} \phi (\text{TurnPrepReturn } (\text{Prepare } d \text{ fp } ap)) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Prepare } d \text{ fp } ap) \\ \phi (\text{TurnPrepReturn } (\text{StepPrepare } d \text{ fp1 } ap1 \text{ fp2 } ap2)) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{StepPrepare } d \text{ fp1 } ap1 \text{ fp2 } ap2) \\ \phi (\text{TurnMiddleReturn } (\text{Pirouette } lp \text{ ap})) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Pirouette } lp \text{ ap}) \\ \phi (\text{TurnMiddleReturn } (\text{Fouette } lp \text{ ap})) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Fouette } lp \text{ ap}) \\ \phi (\text{TurnMiddleReturn } (\text{Pique } lp \text{ ap})) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Pique } lp \text{ ap}) \\ \phi (\text{TurnMiddleReturn } (\text{Chaine } lp \text{ ap})) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Chaine } lp \text{ ap}) \\ \phi (\text{LeapPrepReturn } (\text{Chasse } d)) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Chasse } d) \\ \phi (\text{LeapPrepReturn } (\text{StepStep } d)) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{StepStep } d) \\ \phi (\text{LeapMiddleReturn } (\text{Jete } d \text{ lp } ap)) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Jete } d \text{ lp } ap) \\ \phi (\text{LeapMiddleReturn } (\text{Switch } d \text{ lp } ap)) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Switch } d \text{ lp } ap) \\ \phi (\text{LeapMiddleReturn } (\text{CenterLeap } d \text{ lp } ap)) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{CenterLeap } d \text{ lp } ap) \\ \phi (\text{EndReturn } (\text{Land } fp \text{ ap})) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Land } fp \text{ ap}) \\ \phi (\text{BattementBegEndReturn } (\text{Step } d)) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Step } d) \\ \phi (\text{BattementMiddleReturn } (\text{Grande } d \text{ lp } ap)) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Grande } d \text{ lp } ap) \\ \phi (\text{BattementMiddleReturn } (\text{Petit } d \text{ lp } ap)) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Petit } d \text{ lp } ap) \\ \phi (\text{RondDeJambeReturn } (\text{RondDeJambe } d \text{ ap1 } fp \text{ ap2})) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{RondDeJambe } d \text{ ap1 } fp \text{ ap2}) \\ \phi (\text{FillerReturn } (\text{Walk})) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Walk}) \\ \phi (\text{FillerReturn } (\text{Hold})) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Hold}) \\ \phi (\text{FillerReturn } (\text{Continue})) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Continue}) \end{aligned}$$

instance Algebra TurnPrep MovEnvErr where

$$\begin{aligned} \phi (\text{Prepare } d \text{ fp } ap) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Prepare } d \text{ fp } ap) \\ \phi (\text{StepPrepare } d \text{ fp1 } ap1 \text{ fp2 } ap2) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{StepPrepare } d \text{ fp1 } ap1 \text{ fp2 } ap2) \end{aligned}$$

instance Algebra TurnMiddle MovEnvErr where

$$\begin{aligned} \phi (\text{Pirouette } lp \text{ ap}) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Pirouette } lp \text{ ap}) \\ \phi (\text{Fouette } lp \text{ ap}) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Fouette } lp \text{ ap}) \\ \phi (\text{Pique } lp \text{ ap}) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Pique } lp \text{ ap}) \\ \phi (\text{Chaine } lp \text{ ap}) &= \text{return } \$ \uparrow \$ \text{ toMovementLang } (\text{Chaine } lp \text{ ap}) \end{aligned}$$

instance Algebra LeapPrep MovEnvErr where

ϕ (*Chasse* d) = return $\$ \uparrow \$$ toMovementLang (*Chasse* d)
 ϕ (*StepStep* d) = return $\$ \uparrow \$$ toMovementLang (*StepStep* d)

instance Algebra LeapMiddle MovEnvErr where

ϕ (*Jete* d lp ap) = return $\$ \uparrow \$$ toMovementLang (*Jete* d lp ap)
 ϕ (*Switch* d lp ap) = return $\$ \uparrow \$$ toMovementLang (*Switch* d lp ap)
 ϕ (*CenterLeap* d lp ap) = return $\$ \uparrow \$$ toMovementLang (*CenterLeap* d lp ap)

instance Algebra End MovEnvErr where

ϕ (*Land* fp ap) = return $\$ \uparrow \$$ toMovementLang (*Land* fp ap)

instance Algebra BattementBegEnd MovEnvErr where

ϕ (*Step* d) = return $\$ \uparrow \$$ toMovementLang (*Step* d)

instance Algebra BattementMiddle MovEnvErr where

ϕ (*Grande* d lp ap) = return $\$ \uparrow \$$ toMovementLang (*Grande* d lp ap)
 ϕ (*Petit* d lp ap) = return $\$ \uparrow \$$ toMovementLang (*Petit* d lp ap)

instance Algebra RondDeJambe MovEnvErr where

ϕ (*RondDeJambe* d $ap1$ fp $ap2$) = return $\$ \uparrow \$$ toMovementLang (*RondDeJambe* d $ap1$ fp $ap2$)

instance Algebra Filler MovEnvErr where

ϕ (*Walk*) = return $\$ \uparrow \$$ toMovementLang (*Walk*)
 ϕ (*Hold*) = return $\$ \uparrow \$$ toMovementLang (*Hold*)
 ϕ (*Continue*) = return $\$ \uparrow \$$ toMovementLang (*Continue*)

instance Algebra Continuation MovEnvErr where

ϕ (*Follows* m_1 m_2) =
 do { $m1' \leftarrow m_1$
 ; $m2' \leftarrow m_2$
 ; **case** $m1'$ **of**
 (*MovementVal* m_1) \rightarrow
 case $m2'$ **of**
 (*MovementVal* m_2) \rightarrow return $\$ \uparrow \$$ toMovementLang (*Follows* m_1 m_2)
 _ \rightarrow throwError (Err ("Not a movement"))
 _ \rightarrow throwError (Err ("Not a movement"))
 }

ϕ (*While* m_1 m_2) =
 do { $m1' \leftarrow m_1$
 ; $m2' \leftarrow m_2$

```

; case m1' of
  (MovementVal m1) →
    case m2' of
      (MovementVal m2) → return $ ↑ $ toMovementLang (While m1 m2)
      _ → throwError (Err ("Not a movement"))
    _ → throwError (Err ("Not a movement"))
  }

```

```

instance Algebra VarTerm MovEnvErr where
  φ (Var s) = do { val ← asks (lookupEnv s)
                ; case val of
                  (Just v) → return $ ↑ $ v
                  Nothing → throwError (Err ("Variable " ++ s ++ " not found"))
                }
  where lookupEnv :: String → Env → (Maybe Value)
        lookupEnv s e = lookup s e

```

```

instance Algebra FcnTerm MovEnvErr where
  φ (Lambda s body) =
    do { env ← ask
        ; return $ LambdaVal (λv → (do { v' ← sequence v
                                       ; (local (const ((zip s v') ++ env)) body)
                                     }
                                )
        )
    }

```

```

φ (App e1 e2) = do { e1' ← e1
                  ; case (↓ e1') of
                    (Just (LambdaVal f)) → (f e2)
                    (Just (MovementVal x)) → throwError (Err (((show x) ++ " is not a function")))
                    _ → throwError (Err ("No closure"))
                  }

```

```

evalD :: MovementLang → MovEnvErr
evalD = cata

```

```

evalEnv :: Env → MovementLang → Value
evalEnv env term = ((runReader (cata term)) env)

```

```

initenv :: Env
initenv = [(s, (evalEnv [] v)) | (s, v) ← movements]

execute term = case (evalEnv [(s, (evalEnv initenv v)) | (s, v) ← movements] term) of
  MovementVal x → show x
  LambdaVal x → throwError (Err ("Lambda value returned rather than ..."))

addEnv :: String → MovementLang → Env → Env
addEnv s m e = (s, m) : e

identity :: MovementLang =
  (toMovementLang (Lambda ["x"] (toMovementLang (Var "x"))))

example1 :: MovementLang =
  (toMovementLang
    (App (toMovementLang
      (Lambda ["y", "x"]
        (toMovementLang
          (Follows
            (toMovementLang (Var "y"))
            (toMovementLang (Var "x"))))))))
    [(toMovementLang (Var "RightPrepare")),
     (toMovementLang (Var "PassePirouette"))])

example2 :: MovementLang =
  (toMovementLang
    (App
      (toMovementLang (Var "GenFollows"))
      [(toMovementLang (Var "RightPrepare")),
       (toMovementLang (Var "PassePirouette"))]))

{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances -fallow-overlapping-instances #-}

module Dance2Environment where
import Dance2AST

data Bind a = Bind (String, a)
  deriving (Eq, Show)

```

```

("LeftChaine", (toMovementLang
  (Follows
    (toMovementLang
      (Prepare Dance2AST.Left FourthF FourthA))
    (toMovementLang
      (Follows
        (toMovementLang
          (Chaine Straight FirstA))
        (toMovementLang
          (Land SecondF LowV)))))),
("RightJeteLeap", (toMovementLang
  (Follows
    (toMovementLang
      (StepStep Dance2AST.Right))
    (toMovementLang
      (Follows
        (toMovementLang
          (Jete Dance2AST.Right Developpe HighV))
        (toMovementLang
          (Land FourthF FirstA)))))),
("LeftJeteLeap", (toMovementLang
  (Follows
    (toMovementLang
      (StepStep Dance2AST.Left))
    (toMovementLang
      (Follows
        (toMovementLang
          (Jete Dance2AST.Left Developpe HighV))
        (toMovementLang
          (Land FourthF FirstA)))))),
("RightSwitchLeap", (toMovementLang
  (Follows
    (toMovementLang
      (Chasse Dance2AST.Right))
    (toMovementLang
      (Follows
        (toMovementLang
          (Switch Dance2AST.Right Straight FourthA))
        (toMovementLang
          (Land FourthF FirstA)))))),

```

```

("LeftSwitchLeap", (toMovementLang
  (Follows
    (toMovementLang
      (Chasse Dance2AST.Left))
    (toMovementLang
      (Follows
        (toMovementLang
          (Switch Dance2AST.Left Straight FourthA))
        (toMovementLang
          (Land FourthF FirstA))))))),
("RightCenterLeapLeap", (toMovementLang
  (Follows
    (toMovementLang
      (Chasse Dance2AST.Right))
    (toMovementLang
      (Follows
        (toMovementLang
          (CenterLeap Dance2AST.Right Straight SecondA))
        (toMovementLang
          (Land FourthF LowV))))))),
("LeftCenterLeapLeap", (toMovementLang
  (Follows
    (toMovementLang
      (Chasse Dance2AST.Left))
    (toMovementLang
      (Follows
        (toMovementLang
          (CenterLeap Dance2AST.Left Straight SecondA))
        (toMovementLang (Land FourthF LowV))))))),
("LeftGrandeBattement", (toMovementLang
  (Follows
    (toMovementLang
      (Step Dance2AST.Right))
    (toMovementLang
      (Follows
        (toMovementLang
          (Grande Dance2AST.Left Developpe LDiagona))
        (toMovementLang
          (Step Dance2AST.Left))))))),
("RightGrandeBattement", (toMovementLang

```

```

(Follows
  (toMovementLang
    (Step Dance2AST.Left))
  (toMovementLang
    (Follows
      (toMovementLang
        (Grande Dance2AST.Right Developé RDiagonal
          (toMovementLang
            (Step Dance2AST.Right))))))),
("LeftPetitBattement", (toMovementLang
  (Follows
    (toMovementLang
      (Step Dance2AST.Right))
    (toMovementLang
      (Follows
        (toMovementLang
          (Petit Dance2AST.Left Developé LDiagonal
            (toMovementLang
              (Step Dance2AST.Left))))))),
("RightPetitBattement", (toMovementLang
  (Follows
    (toMovementLang
      (Step Dance2AST.Left))
    (toMovementLang
      (Follows
        (toMovementLang
          (Petit Dance2AST.Right Developé RDiagonal
            (toMovementLang
              (Step Dance2AST.Right))))))),
("GenFollows", (toMovementLang
  (Lambda ["y", "x"]
    (toMovementLang
      (Follows
        (toMovementLang (Var "y"))
        (toMovementLang (Var "x")))))))]

```