

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 represents current thoughts on the specification of data structures and functions for music and dance. It is intended to serve two purposes: (i) provide the beginnings of a technical report; and (ii) demonstrate literate Haskell programming. Literate Haskell programming combines documentation in \LaTeX form and Haskell code into a single file. This file can either be loaded directly into the GHCi or HUGS environment like traditional source code, or run through the `lhs2TeX` program to generate a documentation file that can be used with \LaTeX to generate a PDF or PostScript documentation file. This is a very cool thing!!

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 $\phi a \rightarrow a$. $map_f cata$ 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, $map_f cata$ thus maps *cata* over *out* extracts the argument to *Rec f*.

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

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.

```

module DanceAST where

```

```

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

data BeatStructure =
  Beat |
  Pair BeatStructure |
  Quad BeatStructure |
  Octet BeatStructure
  deriving (Eq, Show)

```

The *BeatStructure* datatype derives functions in classes *Eq* and *Show* to allow comparison and display of *BeatStructure* concepts.

The *duration* function transforms a *BeatStructure* into an *Integer* value indicating its duration in beats. This function is easily defined by providing a case for each entry in the *BeatStructure* datatype. Each case in the *duration* function is simply an equation that defines the *duration* of one possible *BeatStructure*. This is exactly what you would do if you were defining a mathematical model.

```

duration :: BeatStructure → Integer
duration Beat = 1
duration (Pair x) = 2 * duration x
duration (Quad x) = 4 * duration x
duration (Octet x) = 8 * duration x

```

Note that parenthesis are placed around patterns of length greater than 1. Haskell needs this to parse the datatype properly.

After you load the module into GHCI, you can type the following at the command line to test the *duration* function:

- *duration Beat*
- *duration (Pair Beat)*
- *duration (Quad (Pair Beat))*
- *duration (Octet (Pair (Quad Beat)))*

2.1 The Movement Type

The most basic movements in dance can be done on either side of the body or to the center. The datatype *Direction* defines the three sides movements can be performed on: left, right, or center.

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

The building blocks of dance are the five positions. These positions can be applied to the feet and the arms. The positions of the feet are always the same, however the positioning of the arms can vary depending on which method you were taught. Below are the five basic positions I will assume throughout this program.

First position: both heels together, toes turned out like a penguin, arms resting in front of the hips, with fingertips curved towards one another

Second position: toes turned out like a penguin, heels shoulder width apart, arms extended to the sides (in the shape of a T)

Third position: toes turned out like a penguin, front heel resting in the arch of the back foot, arm correlating with front foot extended to side (half of a T), opposite arm curved over the head

Fourth position: toes turned out like a penguin, front heel in line with back foot's toe, about a foot of space between the feet, arm correlating with front foot extended to side (half of T), opposite arm curved in front of body.

Fifth position: toes turned out like a penguin, front heel touching back toe, and back toe touching front heel, arms curved above head with fingertips pointing towards one another

All five positions can be done on either side of the body. For example you could perform third position on the left side, therefore meaning that your left heel would be in the arch of the right and your left arm would be extended to the side, while the right arm would be curved over the top of the head.

```
data FootPosition =  
  FirstF |
```

```

    SecondF |
    ThirdF |
    FourthF |
    FifthF |
    PrevF |
    NextF
deriving (Eq, Show)

```

Like the feet, the arms have positions too. Below are the descriptions of the different arm positions.

First Position involves the arms resting in front of the hips with the fingertips curved towards each other.

Second Position has the arms stretched out to the sides in the shape of a T.

Third Position can vary depending on the style you were taught. For the purposes of this program we will assume that Third Position has the arm corresponding with the front foot in seconds while the opposite arm is curved above the head.

Fourth Position involves the arm corresponding to the front foot in second, while the opposite arm is curved in front of the body.

Fifth Position has both arms above the head with fingertips curved towards one another.

```

data ArmPosition =
    FirstA |
    SecondA |
    ThirdA |
    FourthA |
    FifthA |
    HighV |
    LowV |
    Tee |
    RDiagonal |
    LDiagonal |
    PrevA |
    NextA
deriving (Eq, Show)

```

To define most movements you must define what position the legs are in. Below are a list of leg positions I will be using throughout this program. Attitude is a position where the leg is lifted to the back of the body, the knee is bent at an angle of 90 degrees and well turned out so that the knee is higher than the foot. Coupe involves the working foot being placed on the part of the leg between the base of the calf and the beginning of the ankle. Passe takes the foot of the working leg and places it at the knee of the supporting leg, with both knees facing forward. Open Passe is the same as Passe except the working knee is turned out to the side. Develope is a movement in which the working leg is drawn up to the knee of the supporting leg and slowly extended to an open position. Straight is the opposite of Develope, where when the leg is extended, the knee does not bend.

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

All turns must begin with a preparation or a *TurnBeg*. Below are two types of preparations that will be used. Prepare simply takes one of the five positions (most frequently fourth) and a direction the turn will be executed to. StepPrepare takes two positions (most frequently second followed by fourth) and a direction the turn will be executed to.

```
data TurnBeg =
  Prepare Direction FootPosition ArmPosition |
  StepPrepare Direction FootPosition ArmPosition FootPosition ArmPosition
deriving (Eq, Show)
```

TurnMiddle defines the placement of the working leg during a turn. Although we won't be able to represent this without some sort of graphics, I went ahead and included it so that there could be some variety to turns. Also, this will come in handy when it's time to write the type checker because there are certain leg positions that are available, but that you wouldn't use during a turn.

```

data TurnMiddle =
  TurnLeg LegPosition ArmPosition
deriving (Eq, Show)

```

All turns must also have an ending which is specified with a position (most frequently second position).

```

data TurnEnd =
  TurnEndPos FootPosition ArmPosition
deriving (Eq, Show)

```

A *Turn* is specified by a preparation, *TurnBeg*, a position for the working leg, *TurnMiddle*, and an ending, *TurnEnd*, as well as a duration. Right now I have this defined using *BeatStructure*, but eventually I think we're going to want to write functions to create a double turn, triple turn, etc. But for now we'll just use *BeatStructure*.

```

data Spin =
  Pirouette TurnMiddle |
  Fouette TurnMiddle |
  Chaine TurnMiddle |
  Pique TurnMiddle
deriving (Eq, Show)

```

Like a *Turn* a leap must have a beginning also. To prepare for a leap you can either *Chasse* which is a step-together-step motion where the feet must touch each other on the "together" part. Or the leap can begin with just two steps leading into it, the *StepStep* option. A leap can only last for one count since you can't hover in the air, so no time duration is specified here.

```

data LeapBeg =
  Chasse Direction |
  StepStep Direction
deriving (Eq, Show)

```

When doing a *Leap* the front and back legs can be doing the same thing, or they can be doing different things. Actually, now that I'm thinking about it, no they can't. I'm not sure why I thought I needed to distinguish between the front and back legs of a leap, but now I'm thinking I don't need to. I'm going to leave it for now, but will probably end up changing this structure a bit.

```
data FrontLeg =  
  FrontLeapLeg LegPosition  
  deriving (Eq, Show)
```

```
data BackLeg =  
  BackLeapLeg LegPosition ArmPosition  
  deriving (Eq, Show)
```

There must be an ending to a leap. You must complete the move before you can move on to another. The usual ending for a leap is fourth position. Therefore, in the code this structure is defined by a position.

```
data LeapEnd =  
  LeapEndPos FootPosition ArmPosition  
  deriving (Eq, Show)
```

There are many different types of leaps, but for now I have defined three. The regular jete, is with either the right or left leg in front and the opposite in the back. Essentially you are doing the splits in the air. The switch leap is like a jete, except while you are in mid air the leg positions are switched. Therefore, if you leave the group with your right leg leading, while in the air you will switch to where your left leg is leading, or vice versa. The center leap is a lead with the torso facing the audience and the legs to either side of the body. It looks as if the dancer is doing the middle splits while in the air.

```
data Jump =  
  Jete FrontLeg BackLeg |  
  Switch FrontLeg BackLeg |  
  CenterLeap FrontLeg BackLeg  
  deriving (Eq, Show)
```

A *Battement* is a kick of the leg. Either a Grand one, high, or a petit one, low. As the kicking leg leaves the ground it can either come up straight or developpe, this is why the structure takes a *LegPosition*.

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

```

data BattementMiddle =
  BM Direction LegPosition ArmPosition
  deriving (Eq, Show)

```

```

data Battement =
  Grande BattementMiddle |
  Petit BattementMiddle
  deriving (Eq, Show)

```

Fill defines what will be happening between the major movements. As of now, I'm just considering it walking across the floor.

```

data Filler =
  Walk FootPosition BeatStructure FootPosition |
  Hold |
  Continue
  deriving (Eq, Show)

```

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.

```

data Movement = TurnPrep TurnBeg
  | Turn Spin
  | TurnLand TurnEnd
  | LeapPrep LeapBeg
  | Leap Jump
  | LeapLand LeapEnd
  | KickPrepLand BattementBegEnd
  | Kick Battement
  | Fill Filler
  | Follows Movement Movement
  | While Movement Movement
  deriving (Eq, Show)

```

–Parser Data Structures

```

data ProgramRet
  = P RNRet StyleRet Integer DescripRet [EightCount] DescripRet

```

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 ReturnType =  
  ParserReturn Movement  
deriving (Eq, Show)
```

```
type EightCount = (Integer, [TimedMovement])
```

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

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

```
type TimedMovement = (MCountRet, ReturnType)
```

```
{-# OPTIONS -fglasgow-exts #-}
```

```
module DanceTypes where
```

```
import DanceAST
```

Typeof

```
class Typed a b | a → b where  
  typeofD :: a → b
```

PosTy

```
data PosTy = FootPosTy  
  | ArmPosTy  
  | DirectionTy  
  | LegPosTy  
  deriving (Eq, Show)
```

PrepTy

```
data PrepTy = PrepareTy Direction FootPosition ArmPosition  
  | StepPrepareTy Direction FootPosition ArmPosition FootPosition ArmPosition  
  | ChasseTy Direction  
  | StepStepTy Direction  
  | StepTy Direction  
  deriving (Eq, Show)
```

Converting all Preps to Types

```
instance Typed TurnBeg PrepTy where  
  typeofD (Prepare d f a) = (PrepareTy d f a)  
  typeofD (StepPrepare d f1 a1 f2 a2) = (StepPrepareTy d f1 a1 f2 a2)
```

```
instance Typed LeapBeg PrepTy where  
  typeofD (Chasse d) = (ChasseTy d)  
  typeofD (StepStep d) = (StepStepTy d)
```

```
instance Typed BattementBegEnd PrepTy where  
  typeofD (Step d) = (StepTy d)
```

MiddleTy

```
data MiddleTy = TurnLegTy LegPosition ArmPosition
              | FrontLegTy LegPosition
              | BackLegTy LegPosition ArmPosition
              | BMTy Direction LegPosition ArmPosition
deriving (Eq, Show)
```

Converting all Middles to Types

```
instance Typed TurnMiddle MiddleTy where
  typeof $\mathcal{D}$  (TurnLeg l a) = (TurnLegTy l a)
```

```
instance Typed FrontLeg MiddleTy where
  typeof $\mathcal{D}$  (FrontLeapLeg l) = (FrontLegTy l)
```

```
instance Typed BackLeg MiddleTy where
  typeof $\mathcal{D}$  (BackLeapLeg l a) = (BackLegTy l a)
```

```
instance Typed BattementMiddle MiddleTy where
  typeof $\mathcal{D}$  (BM d l a) = (BMTy d l a)
```

EndTy

```
data EndTy = TurnEndPosTy FootPosition ArmPosition
           | LeapEndPosTy FootPosition ArmPosition
deriving (Eq, Show)
```

Converting all Ends to Types

```
instance Typed TurnEnd EndTy where
  typeof $\mathcal{D}$  (TurnEndPos f a) = (TurnEndPosTy f a)
```

```
instance Typed LeapEnd EndTy where
  typeof $\mathcal{D}$  (LeapEndPos f a) = (LeapEndPosTy f a)
```

BeatStructureTy

```
data BeatStructureTy = BeatTy
deriving (Eq, Show)
```

—————Converting BeatStructure to a Type—————

```
instance Typed BeatStructure BeatStructureTy where  
  typeofD Beat = BeatTy
```

—————Ty—————

```
data TyD = TyFA (FootPosition, ArmPosition) (FootPosition, ArmPosition)  
  | TyLA (LegPosition, ArmPosition) (LegPosition, ArmPosition)  
  | TyL (LegPosition, LegPosition)  
  | TyFALA (FootPosition, ArmPosition) (LegPosition, ArmPosition)  
  | TyLAFA (LegPosition, ArmPosition) (FootPosition, ArmPosition)  
  | TyLFA (LegPosition) (FootPosition, ArmPosition)  
  | TyFAL (FootPosition, ArmPosition) (LegPosition)  
  | TyLLA (LegPosition) (LegPosition, ArmPosition)  
  | TyLAL (LegPosition, ArmPosition) (LegPosition)  
deriving (Eq, Show)
```

```
something :: MovementTy → Type  
something m = ((initialPos m) (finalPos m))
```

—————MovementTy—————

```
data MovementTy = TurnPrepTy PrepTy  
  | TurnTy MovementTy  
  | TurnLandTy EndTy  
  | LeapPrepTy PrepTy  
  | LeapTy MovementTy  
  | LeapLandTy EndTy  
  | KickPrepTy PrepTy  
  | KickTy MovementTy  
  | PirouetteTy MiddleTy  
  | FouetteTy MiddleTy  
  | PiqueTy MiddleTy  
  | ChaineTy MiddleTy  
  | JeteTy MiddleTy MiddleTy  
  | SwitchTy MiddleTy MiddleTy  
  | CenterLeapTy MiddleTy MiddleTy  
  | GrandeTy MiddleTy
```

```

| PetitTy MiddleTy
| FillTy MovementTy
| WalkTy Direction BeatStructureTy Direction
| HoldTy
| ContinueTy
| FolowsTy MovementTy MovementTy
| WhileTy MovementTy MovementTy
| Type TyD
deriving (Eq, Show)

```

—typeof definitions—

instance *Typed Spin MovementTy where*

```

typeofD (Pirouette tm) = (PirouetteTy (typeofD tm))
typeofD (Fouette tm) = (FouetteTy (typeofD tm))
typeofD (Chaine tm) = (ChaineTy (typeofD tm))
typeofD (Pique tm) = (PiqueTy (typeofD tm))

```

instance *Typed Jump MovementTy where*

```

typeofD (Jete f b) = (JeteTy (typeofD f) (typeofD b))
typeofD (Switch f b) = (SwitchTy (typeofD f) (typeofD b))
typeofD (CenterLeap f b) = (CenterLeapTy (typeofD f) (typeofD b))

```

instance *Typed Battement MovementTy where*

```

typeofD (Grande bm) = (GrandeTy (typeofD bm))
typeofD (Petit bm) = (PetitTy (typeofD bm))

```

instance *Typed Filler MovementTy where*

```

-- typeof (Walk f b f2) = (WalkTy (typeof f) (typeof b) (typeof f2))
typeofD Hold = HoldTy
typeofD Continue = ContinueTy

```

instance *Typed Movement MovementTy where*

```

typeofD (TurnPrep tb) = (TurnPrepTy (typeofD tb))
typeofD (Turn s) = (TurnTy (typeofD s))
typeofD (TurnLand te) = (TurnLandTy (typeofD te))
typeofD (LeapPrep lb) = (LeapPrepTy (typeofD lb))
typeofD (Leap j) = (LeapTy (typeofD j))
typeofD (LeapLand le) = (LeapLandTy (typeofD le))

```

```

typeofℳ (KickPrepLand bbe) = (KickPrepTy (typeofℳ bbe))
typeofℳ (Kick b) = (KickTy (typeofℳ b))
typeofℳ (Fill f) = (FillTy (typeofℳ f))
typeofℳ (Follows m1 m2) = (FollowsTy (typeofℳ m1) (typeofℳ m2))
typeofℳ (While m1 m2) = (WhileTy (typeofℳ m1) (typeofℳ m2))

```

- Defining initial and
final position functions for Types

class *Positionable* a b **where**

initialPos :: a → b

finalPos :: a → b

instance *Positionable* *PrepTy* (*FootPosition*, *ArmPosition*) **where**

initialPos (*PrepareTy* _ f a) = (f, a)

initialPos (*StepPrepareTy* _ f a _ _) = (f, a)

initialPos (*ChasseTy* *DanceAST.Right*) = (*FourthF*, *FourthA*)

initialPos (*ChasseTy* *DanceAST.Left*) = (*FourthF*, *FourthA*)

initialPos (*ChasseTy* *Center*) = (*SecondF*, *SecondA*)

initialPos (*StepStepTy* _) = (*FourthF*, *FirstA*)

initialPos (*StepTy* _) = (*ThirdF*, *FirstA*)

finalPos (*ChasseTy* *DanceAST.Right*) = (*FourthF*, *FourthA*)

finalPos (*ChasseTy* *DanceAST.Left*) = (*FourthF*, *FourthA*)

finalPos (*ChasseTy* *Center*) = (*SecondF*, *SecondA*)

finalPos (*StepStepTy* _) = (*FourthF*, *FirstA*)

finalPos (*PrepareTy* _ f a) = (f, a)

finalPos (*StepPrepareTy* _ _ _ f a) = (f, a)

finalPos (*StepTy* _) = (*ThirdF*, *FirstA*)

instance *Positionable* *MiddleTy* (*LegPosition*, *ArmPosition*) **where**

initialPos (*TurnLegTy* l a) = (l, a)

initialPos (*BackLegTy* l a) = (l, a)

initialPos (*BMTy* _ l a) = (l, a)

finalPos (*TurnLegTy* l a) = (l, a)

finalPos (*BackLegTy* l a) = (l, a)

finalPos (*BMTy* _ l a) = (l, a)

instance *Positionable* *MiddleTy* *LegPosition* **where**

initialPos (*FrontLegTy* l) = l

finalPos (*FrontLegTy* l) = l

instance *Positionable EndTy* (*FootPosition*, *ArmPosition*) **where**

initialPos (*TurnEndPosTy* *f a*) = (*f*, *a*)
initialPos (*LeapEndPosTy* *f a*) = (*f*, *a*)
finalPos (*TurnEndPosTy* *f a*) = (*f*, *a*)
finalPos (*LeapEndPosTy* *f a*) = (*f*, *a*)

instance *Positionable MovementTy* (*LegPosition*, *ArmPosition*) **where**

initialPos (*PirouetteTy* *m*) = *initialPos* *m*
initialPos (*FouetteTy* *m*) = *initialPos* *m*
initialPos (*ChaineTy* *m*) = *initialPos* *m*
initialPos (*PiqueTy* *m*) = *initialPos* *m*
initialPos (*GrandeTy* *m*) = *initialPos* *m*
initialPos (*PetitTy* *m*) = *initialPos* *m*
finalPos (*PirouetteTy* *m*) = *finalPos* *m*
finalPos (*FouetteTy* *m*) = *finalPos* *m*
finalPos (*ChaineTy* *m*) = *finalPos* *m*
finalPos (*PiqueTy* *m*) = *finalPos* *m*
finalPos (*GrandeTy* *m*) = *finalPos* *m*
finalPos (*PetitTy* *m*) = *finalPos* *m*

instance *Positionable MovementTy* (*FootPosition*, *ArmPosition*) **where**

initialPos (*JeteTy* _ _) = (*PrevF*, *PrevA*)
initialPos (*SwitchTy* _ _) = (*PrevF*, *PrevA*)
initialPos (*CenterLeapTy* _ _) = (*PrevF*, *PrevA*)
initialPos (*WalkTy* _ _ _) = (*PrevF*, *PrevA*)
initialPos *HoldTy* = (*PrevF*, *PrevA*)
initialPos *ContinueTy* = (*PrevF*, *PrevA*)
initialPos (*TurnPrepTy* *tp*) = *initialPos* *tp*
initialPos (*TurnLandTy* *tl*) = *initialPos* *tl*
initialPos (*LeapPrepTy* *lp*) = *initialPos* *lp*
initialPos (*LeapLandTy* *ll*) = *initialPos* *ll*
initialPos (*FollowsTy* *m*₁ _) = *initialPos* *m*₁
initialPos (*WhileTy* *m*₁ _) = *initialPos* *m*₁
finalPos (*JeteTy* _ _) = (*NextF*, *NextA*)
finalPos (*SwitchTy* _ _) = (*NextF*, *NextA*)
finalPos (*CenterLeapTy* _ _) = (*NextF*, *NextA*)
finalPos (*WalkTy* _ _ _) = (*NextF*, *NextA*)
finalPos *HoldTy* = (*NextF*, *NextA*)
finalPos *ContinueTy* = (*NextF*, *NextA*)

$$\begin{aligned}
\text{finalPos } (\text{TurnPrepTy } tp) &= \text{finalPos } tp \\
\text{finalPos } (\text{TurnLandTy } tl) &= \text{finalPos } tl \\
\text{finalPos } (\text{LeapPrepTy } lp) &= \text{finalPos } lp \\
\text{finalPos } (\text{LeapLandTy } ll) &= \text{finalPos } ll \\
\text{finalPos } (\text{FollowsTy } _ m_2) &= \text{finalPos } m_2 \\
\text{finalPos } (\text{WhileTy } _ m_2) &= \text{finalPos } m_2
\end{aligned}$$

—Defining initial and final position functions for Movements

instance *Positionable TurnBeg* (*FootPosition, ArmPosition*) **where**

$$\begin{aligned}
\text{initialPos } (\text{Prepare } _ f a) &= (f, a) \\
\text{initialPos } (\text{StepPrepare } _ f a _ _) &= (f, a) \\
\text{finalPos } (\text{Prepare } _ f a) &= (f, a) \\
\text{finalPos } (\text{StepPrepare } _ _ _ f a) &= (f, a)
\end{aligned}$$

instance *Positionable TurnMiddle* (*LegPosition, ArmPosition*) **where**

$$\begin{aligned}
\text{initialPos } (\text{TurnLeg } l a) &= (l, a) \\
\text{finalPos } (\text{TurnLeg } l a) &= (l, a)
\end{aligned}$$

instance *Positionable TurnEnd* (*FootPosition, ArmPosition*) **where**

$$\begin{aligned}
\text{initialPos } (\text{TurnEndPos } f a) &= (f, a) \\
\text{finalPos } (\text{TurnEndPos } f a) &= (f, a)
\end{aligned}$$

instance *Positionable Spin* (*LegPosition, ArmPosition*) **where**

$$\begin{aligned}
\text{initialPos } (\text{Pirouette } m) &= \text{initialPos } m \\
\text{initialPos } (\text{Fouette } m) &= \text{initialPos } m \\
\text{initialPos } (\text{Chaine } m) &= \text{initialPos } m \\
\text{initialPos } (\text{Pique } m) &= \text{initialPos } m \\
\text{finalPos } (\text{Pirouette } m) &= \text{finalPos } m \\
\text{finalPos } (\text{Fouette } m) &= \text{finalPos } m \\
\text{finalPos } (\text{Chaine } m) &= \text{finalPos } m \\
\text{finalPos } (\text{Pique } m) &= \text{finalPos } m
\end{aligned}$$

instance *Positionable LeapBeg* (*FootPosition, ArmPosition*) **where**

$$\begin{aligned}
\text{initialPos } (\text{Chasse DanceAST.Right}) &= (\text{FourthF}, \text{FourthA}) \\
\text{initialPos } (\text{Chasse DanceAST.Left}) &= (\text{FourthF}, \text{FourthA}) \\
\text{initialPos } (\text{Chasse Center}) &= (\text{SecondF}, \text{SecondA}) \\
\text{initialPos } (\text{StepStep } _) &= (\text{FourthF}, \text{FirstA})
\end{aligned}$$

$finalPos (Chasse\ DanceAST.Right) = (FourthF, FourthA)$
 $finalPos (Chasse\ DanceAST.Left) = (FourthF, FourthA)$
 $finalPos (Chasse\ Center) = (SecondF, SecondA)$
 $finalPos (StepStep _) = (FourthF, FirstA)$

instance *Positionable FrontLeg LegPosition* **where**

$initialPos (FrontLeapLeg\ l) = l$
 $finalPos (FrontLeapLeg\ l) = l$

instance *Positionable BackLeg (LegPosition, ArmPosition)* **where**

$initialPos (BackLeapLeg\ l\ a) = (l, a)$
 $finalPos (BackLeapLeg\ l\ a) = (l, a)$

instance *Positionable LeapEnd (FootPosition, ArmPosition)* **where**

$initialPos (LeapEndPos\ f\ a) = (f, a)$
 $finalPos (LeapEndPos\ f\ a) = (f, a)$

instance *Positionable Jump (FootPosition, ArmPosition)* **where**

$initialPos (Jete\ _ _) = (PrevF, PrevA)$
 $initialPos (Switch\ _ _) = (PrevF, PrevA)$
 $initialPos (CenterLeap\ _ _) = (PrevF, PrevA)$
 $finalPos (Jete\ _ _) = (NextF, NextA)$
 $finalPos (Switch\ _ _) = (NextF, NextA)$
 $finalPos (CenterLeap\ _ _) = (NextF, NextA)$

instance *Positionable BattementBegEnd (FootPosition, ArmPosition)* **where**

$initialPos (Step\ _) = (ThirdF, FirstA)$
 $finalPos (Step\ _) = (ThirdF, FirstA)$

instance *Positionable BattementMiddle (LegPosition, ArmPosition)* **where**

$initialPos (BM\ _ l\ a) = (l, a)$
 $finalPos (BM\ _ l\ a) = (l, a)$

instance *Positionable Battement (LegPosition, ArmPosition)* **where**

$initialPos (Grande\ m) = initialPos\ m$
 $initialPos (Petit\ m) = initialPos\ m$
 $finalPos (Grande\ m) = finalPos\ m$
 $finalPos (Petit\ m) = finalPos\ m$

instance *Positionable Filler (FootPosition, ArmPosition)* **where**

initialPos (*Walk* _ _ _) = (*PrevF*, *PrevA*)
initialPos *Hold* = (*PrevF*, *PrevA*)
initialPos *Continue* = (*PrevF*, *PrevA*)
finalPos (*Walk* _ _ _) = (*NextF*, *NextA*)
finalPos *Hold* = (*NextF*, *NextA*)
finalPos *Continue* = (*NextF*, *NextA*)

instance *Positionable Movement* (*FootPosition*, *ArmPosition*) **where**

initialPos (*TurnPrep* *tp*) = *initialPos* *tp*
initialPos (*TurnLand* *tl*) = *initialPos* *tl*
initialPos (*LeapPrep* *lp*) = *initialPos* *lp*
initialPos (*Leap* *j*) = *initialPos* *j*
initialPos (*LeapLand* *ll*) = *initialPos* *ll*
initialPos (*KickPrepLand* *kp*) = *initialPos* *kp*
initialPos (*Fill* *f*) = *initialPos* *f*
initialPos (*Follows* *m*₁ _) = *initialPos* *m*₁
initialPos (*While* *m*₁ _) = *initialPos* *m*₁
finalPos (*TurnPrep* *tp*) = *finalPos* *tp*
finalPos (*TurnLand* *tl*) = *finalPos* *tl*
finalPos (*LeapPrep* *lp*) = *finalPos* *lp*
finalPos (*Leap* *j*) = *finalPos* *j*
finalPos (*LeapLand* *ll*) = *finalPos* *ll*
finalPos (*KickPrepLand* *kp*) = *finalPos* *kp*
finalPos (*Fill* *f*) = *finalPos* *f*
finalPos (*Follows* _ *m*₂) = *finalPos* *m*₂
finalPos (*While* _ *m*₂) = *finalPos* *m*₂

instance *Positionable Movement* (*LegPosition*, *ArmPosition*) **where**

initialPos (*Turn* *s*) = *initialPos* *s*
initialPos (*Kick* *b*) = *initialPos* *b*
finalPos (*Turn* *s*) = *finalPos* *s*
finalPos (*Kick* *b*) = *finalPos* *b*

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 DanceParser 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 DancePPrintLevel1
import DanceAST

```

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 ProgramRet
programP = do { whiteSpaceP
               ; rn ← routineNameP

```

```
; rs ← routineStyleP
; nod ← numOfDancersP
; bp ← begPosP
; l ← many lineP
; ep ← endPosP
; return (P 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** { *i*₁ ← *intP*
 ; *dashP*
 ; *i*₂ ← *intP*
 ; return (*MCDash* *i*₁ *i*₂) }

mCountInnerComma :: Parser MCountRet
mCountInnerComma = **do** { *i*₁ ← *intP*
 ; *commaP*
 ; *i*₂ ← *intP*
 ; return (*MCComma* *i*₁ *i*₂) }

mCountInnerIntAnd :: Parser MCountRet
mCountInnerIntAnd = **do** { *i*₁ ← *intP*
; *andP*
; return (MCIntAnd *i*₁) }

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

mCountInnerInt :: Parser MCountRet
mCountInnerInt = **do** { *i*₁ ← *intP*
; return (MCInt *i*₁) }

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

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

directionP :: Parser Direction
directionP = **do** { *reservedP* "right"
; return DanceAST.Right }

```
< | >
do { reservedP "left"
    ; return DanceAST.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 }
```

```
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 }

```

```

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
turnMiddleP = do { reservedP "leg"
                  ; equalP
                  ; lp ← legPositionP
                  ; reservedP "arms"
                  ; equalP
                  ; ap ← armPositionP
                  ; return (TurnLeg lp ap) }
```

```
turnEnd1P :: Parser TurnEnd
turnEnd1P = do { optional mCountP
                ; reservedP "land"
                ; reservedP "feet"
                ; equalP
                ; fp ← footPositionP
                ; reservedP "arms"
                ; equalP
                ; ap ← armPositionP
                ; return (TurnEndPos fp ap) }
```

```
turnEndP :: Parser Movement
turnEndP = do { t ← turnEnd1P
               ; return (TurnLand t) }
```

```
turnPrep1P :: Parser TurnBeg
turnPrep1P = 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)}

```

```

turnPrepP :: Parser Movement
turnPrepP = do { t ← turnPrep1P
; return (TurnPrep t)}

```

```

pirouetteP :: Parser Spin
pirouetteP = do { optional mCountP
; reservedP "pirouette"
; tm ← turnMiddleP
; return (Pirouette tm)}

```

```

fouetteP :: Parser Spin

```

```
fouetteP = do { optional mCountP
                ; reservedP "fouette"
                ; tm ← turnMiddleP
                ; return (Fouette tm) }
```

```
chaineP :: Parser Spin
chaineP = do { optional mCountP
                ; reservedP "chaine"
                ; tm ← turnMiddleP
                ; return (Chaine tm) }
```

```
piqueP :: Parser Spin
piqueP = do { optional mCountP
                ; reservedP "pique"
                ; tm ← turnMiddleP
                ; return (Pique tm) }
```

```
turnTypeP :: Parser Spin
turnTypeP = do { p ← try pirouetteP
                ; return p }
< | >
do { f ← try fouetteP
      ; return f }
< | >
do { c ← try chaineP
      ; return c }
< | >
do { p ← try piqueP
      ; return p }
```

turnP :: Parser Movement
turnP = **do** { *tt* ← *turnTypeP*
 ; *return* (Turn *tt*) }

leapPrep1P :: Parser LeapBeg
leapPrep1P = **do** { *optional mCountP*
 ; *reservedP* "chasse"
 ; *d* ← *directionP*
 ; *return* (Chasse *d*) }
< | >
do { *optional mCountP*
 ; *reservedP* "step-step"
 ; *d* ← *directionP*
 ; *return* (StepStep *d*) }

leapPrepP :: Parser Movement
leapPrepP = **do** { *l* ← *leapPrep1P*
 ; *return* (LeapPrep *l*) }

frontLegP :: Parser FrontLeg
frontLegP = **do** { *optional mCountP*
 ; *reservedP* "front"
 ; *reservedP* "leg"
 ; *equalP*
 ; *lp* ← *legPositionP*
 ; *return* (FrontLeapLeg *lp*) }

backLegP :: Parser BackLeg
backLegP = **do** { *reservedP* "back"
 ; *reservedP* "leg"

```
; equalP
; lp ← legPositionP
; reservedP "arms"
; equalP
; ap ← armPositionP
; return (BackLeapLeg lp ap)}
```

```
leapEnd1P :: Parser LeapEnd
leapEnd1P = do { optional mCountP
; reservedP "land"
; reservedP "feet"
; equalP
; fp ← footPositionP
; reservedP "arms"
; equalP
; ap ← armPositionP
; return (LeapEndPos fp ap)}
```

```
leapEndP :: Parser Movement
leapEndP = do { l ← leapEnd1P
; return (LeapLand l)}
```

```
jeteP :: Parser Jump
jeteP = do { optional mCountP
; reservedP "jete"
; fl ← frontLegP
; bl ← backLegP
; return (Jete fl bl)}
```

```
switchP :: Parser Jump
switchP = do { optional mCountP
               ; reservedP "switch"
               ; reservedP "leap"
               ; fl ← frontLegP
               ; bl ← backLegP
               ; return (Switch fl bl) }
```

```
centerLeapP :: Parser Jump
centerLeapP = do { optional mCountP
                   ; reservedP "center"
                   ; reservedP "leap"
                   ; fl ← frontLegP
                   ; bl ← backLegP
                   ; return (CenterLeap fl bl) }
```

```
leapTypeP :: Parser Jump
leapTypeP = do { j ← try jeteP
                 ; return j }
< | >
do { s ← try switchP
     ; return s }
< | >
do { c ← try centerLeapP
     ; return c }
```

```
leapP :: Parser Movement
leapP = do { l ← leapTypeP
             ; return (Leap l) }
```

```
battementBegEnd1P :: Parser BattementBegEnd  
battementBegEnd1P = do { optional mCountP  
    ; reservedP "step"  
    ; d ← directionP  
    ; return (Step d) }
```

```
battementBegEndP :: Parser Movement  
battementBegEndP = do { b ← battementBegEnd1P  
    ; return (KickPrepLand b) }
```

```
battementMiddleP :: Parser BattementMiddle  
battementMiddleP = do { d ← directionP  
    ; reservedP "leg"  
    ; equalP  
    ; lp ← legPositionP  
    ; reservedP "arms"  
    ; equalP  
    ; ap ← armPositionP  
    ; return (BM d lp ap) }
```

```
grandeP :: Parser Battement  
grandeP = do { optional mCountP  
    ; reservedP "grande"  
    ; reservedP "battement"  
    ; bm ← battementMiddleP  
    ; return (Grande bm) }
```

```
petitP :: Parser Battement  
petitP = do { optional mCountP  
    ; reservedP "petit"  
    ; reservedP "battement"  
    ; bm ← battementMiddleP  
    ; return (Petit bm) }
```

kickTypeP :: Parser Battement
kickTypeP = **do** { *g* ← *grandeP*
 ; *return g* }
 < | >
 do { *p* ← *petitP*
 ; *return p* }

kickP :: Parser Movement
kickP = **do** { *m* ← *kickTypeP*
 ; *return (Kick m)* }

fillerP :: Parser Filler
fillerP = **do** { *reservedP* "hold"
 ; *return (Hold)* }

fillP :: Parser Movement
fillP = **do** { *f* ← *fillerP*
 ; *return (Fill f)* }

movementP :: Parser Return Type
movementP = **do** { *m* ← *turnPrepP*
 ; *return (ParserReturn m)* }
 < | >
 do { *m* ← *turnP*
 ; *return (ParserReturn m)* }
 < | >
 do { *m* ← *turnEndP*
 ; *return (ParserReturn m)* }

```

    < | >
  do { m ← leapPrepP
      ; return (ParserReturn m) }
    < | >
  do { m ← leapP
      ; return (ParserReturn m) }
    < | >
  do { m ← leapEndP
      ; return (ParserReturn m) }
    < | >
  do { m ← battementBegEndP
      ; return (ParserReturn m) }
    < | >
  do { m ← kickP
      ; return (ParserReturn m) }
    < | >
  do { m ← fillP
      ; return (ParserReturn 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 (programPP mov)
                  }

```

–Some Functions:

module PrettyPrinter where

```
module DancePPrintLevel1 where  
import PPrint  
    -- import Dance  
import DanceAST  
    -- import DanceParser
```

and get rid of styleInnerPP -- Can combine this into routineStylePP

```
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"
```

```
directionPP :: Direction → Doc  
directionPP DanceAST.Right = text "Right"  
directionPP DanceAST.Left = text "Left"  
directionPP Center = text "Center"
```

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

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"

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"

turnBegPP :: *TurnBeg* → *Doc*
turnBegPP (*Prepare* *d fp ap*) = *text* "Prepare" < + > (*directionPP* *d*) < + > *text* "with ."
turnBegPP (*StepPrepare* *d fp1 ap1 fp2 ap2*) = *text* "Step" < + > (*directionPP* *d*) < + >

turnMiddlePP :: *TurnMiddle* → *Doc*
turnMiddlePP (*TurnLeg* *lp ap*) = *text* "Leg in" < + > (*legPosPP* *lp*) < + > *text* "and arm

turnEndPP :: *TurnEnd* → *Doc*
turnEndPP (*TurnEndPos fp ap*) = *text* "Land in" < + > (*footPosPP fp*) < + > *text* "wit

spinPP :: *Spin* → *Doc*
spinPP (*Pirouette tm*) = (*turnMiddlePP tm*)
spinPP (*Fouette tm*) = (*turnMiddlePP tm*)
spinPP (*Chaine tm*) = (*turnMiddlePP tm*)
spinPP (*Pique tm*) = (*turnMiddlePP tm*)

leapBegPP :: *LeapBeg* → *Doc*
leapBegPP (*Chasse d*) = *text* "Chasse" < + > (*directionPP d*)
leapBegPP (*StepStep d*) = *text* "Step" < + > (*directionPP d*)

frontLegPP :: *FrontLeg* → *Doc*
frontLegPP (*FrontLeapLeg lp*) = *text* "Leap with front leg" < + > (*legPosPP lp*)

backLegPP :: *BackLeg* → *Doc*
backLegPP (*BackLeapLeg lp ap*) = *text* "back leg" < + > (*legPosPP lp*) < + > *text* "wit

leapEndPP :: *LeapEnd* → *Doc*
leapEndPP (*LeapEndPos fp ap*) = *text* "Land with feet in" < + > (*footPosPP fp*) < + >

jumpPP :: *Jump* → *Doc*
jumpPP (*Jete fl bl*) = (*frontLegPP fl*) < + > (*backLegPP bl*)
jumpPP (*Switch fl bl*) = (*frontLegPP fl*) < + > (*backLegPP bl*)
jumpPP (*CenterLeap fl bl*) = (*frontLegPP fl*) < + > (*backLegPP bl*)

battementBegEndPP :: *BattementBegEnd* → *Doc*
battementBegEndPP (*Step d*) = *text "Step"* < + > (*directionPP d*)

battementMiddlePP :: *BattementMiddle* → *Doc*
battementMiddlePP (*BM d lp ap*) = *text "Kick"* < + > (*directionPP d*) < + > *text "leg"*

battementPP :: *Battement* → *Doc*
battementPP (*Grande bm*) = (*battementMiddlePP bm*)
battementPP (*Petit bm*) = (*battementMiddlePP bm*)

fillerPP :: *Filler* → *Doc*
fillerPP (*Walk fp1 bs fp2*) = *text "Start walking from"* < + > (*footPosPP fp1*) < + > *t*
fillerPP Continue = *text "Continue"*
fillerPP Hold = *text "Hold"*

movementPP :: *Movement* → *Doc*
movementPP (*TurnPrep tb*) = (*turnBegPP tb*) <> *linebreak*
movementPP (*Turn s*) = (*spinPP s*) <> *linebreak*
movementPP (*TurnLand te*) = (*turnEndPP te*) <> *linebreak*
movementPP (*LeapPrep lb*) = (*leapBegPP lb*) <> *linebreak*
movementPP (*Leap l*) = (*jumpPP l*) <> *linebreak*
movementPP (*LeapLand le*) = (*leapEndPP le*) <> *linebreak*

```

movementPP (KickPrepLand be) = (battementBegEndPP be) <> linebreak
movementPP (Kick b) = (battementPP b) <> linebreak
movementPP (Fill f) = (fillerPP f) <> linebreak
movementPP (Follows m1 m2) = (movementPP m1) < + > (movementPP m2) <> linebreak
movementPP (While m1 m2) = (movementPP m1) < + > (movementPP m2) <> linebreak

```

```

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 (MCComma 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* → *Doc*
lineMorePP (*mc*, (*ParserReturn ms*)) = (*mCountPP mc*) < + > (*movementPP ms*)

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

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

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