# Haskell and the Arts

How Functional Programmers can
Help, Inspire, or even Be Artists

Paul Hudak
Yale University
Department of Computer Science

QCon San Francisco
November 2008

# Computer Science and Art

- Combinations of Computer Science and some aspect of the Arts has become common at many universities.
- Majors of study are now common in:
  - Video games
  - Computational arts
  - Digital media / multimedia
  - Graphic art
  - Computer music
  - Computer aided design
- In addition, every major art department uses computers in some way for education, creation, and research.

# The Picture at Yale

- New initiative: “Yale C2”
  **C**reative **C**onsilience of Computing and the Arts
- Undergraduate:
  - BS major in *Computing and the Arts*
  - Specialized tracks in Art, Art History, Music, Theater Studies, and (coming soon) Architecture and Film Studies
- Graduate:
  - MS Degree in Computing and the Arts
  - PhD Degree in CS with focus on Computing and the Arts
- New laboratories are also planned

*My goal:*

***Figuring out how PL research can enhance all this.***

# Caveats

- I will raise more questions than I will answer!
  - Examples of work I and others have done.
  - But with a focus on what could be, rather than what is.
- The talk is Haskell- and FP-centric.
  - Feel free to substitute “your favorite language” or “programming paradigm” for “Haskell” or “FP”, respectively, everywhere in this talk.

# How Haskell/FP Could Help Artists

- There is a limitless number of difficult computational problems inspired by the arts:
  - Graphics and animation
  - Modeling and rendering
  - Image processing
  - Audio processing
  - Tools, tools, tools
- The argument for using Haskell/FP in this context is not much different from most other contexts…
- We need the best languages, tools, programming environments, and so on.

# The Sky is the Limit

- Can we create a robotic conductor?
- What does a saxophone the size of a house sound like?
- Can we animate a new choreography?
- Can we create new forms of artistic expression? [see SMule's Ocarina on YouTube!]
- How realistic can a virtual world become?
- Can a computer create an artistic artifact on its own?
  - Or at least "elevator music" or stock graphics design?

# Animusic

- [see video of Pipe Dream on YouTube]
- An example of an application that seems to be begging for FP ideas.
- Combines sophisticated notions of:
  - Physical modeling
  - Graphics and animation
  - Art
  - Music and audio
- Fits in well with Fran, Haskore, Dance, and related ideas (described shortly).

# Can we change the way artists think?

- Three ways that FP can help artists:
  - Abstraction
  - Abstraction
  - Abstraction
- Examples from the Haskell world:
  - The usual: higher-order functions, lazy evaluation, and so on.
  - The unusual: monads, arrows, applicative functors, and other computational abstractions.
- "Monads for Artists"? (yeah right)

# *Should* we change the way artists think?

- Perhaps we *don't* want to change the way artists think!

- Examples:
  - Saying "what" instead of "how". (declarative)
  - Not worrying about resources. (lazy evaluation)
  - No boundaries. (first-class values)
  - Abstracting away detail. (abstraction mechanisms)

- Or perhaps we need to do both:
  - Provide familiar concepts, devoid of irrelevant details.
  - Expose "meta-level" ideas (abstraction techniques!) to allow stretching the imagination.

# Target Audience

- Some artists hate computers.
- Others use them but never look under the hood.
- And some are truly curious, want to know more, are willing to program, explore computer's potential.

attitude

- Some people are left brained.
- Others are right brained.
- And some are both – skilled in logic and intuition.

skill

***We should focus on***
***"curious, ambidextrous-brained people."***

# Haskell and the Arts

- Video games (Frag, Super Nario, …)
- Music (Haskore, HasSound, …)
- Conal Elliott's work on:
  - Fran
  - Pan and Pajama
  - Eros and TV
  - Vertigo

  [see conal.net]

Not a lot…

# Fran, FRP, and Yampa

- FRP = Functional Reactive Programming
- Invented by Conal Elliott
- Became key area of research at Yale:
  - Foundations
  - Implementations
  - Applications:
    - Robotics (both humanoid and mobile)
    - Parallel programming
    - Audio processing / sound synthesis
    - Graphical User Interfaces

# Behaviors in FRP

- Continuous behaviors capture any time-varying quantity, whether:
  - input (sonar, temperature, video, etc.),
  - output (actuator voltage, velocity vector, etc.), or
  - intermediate values internal to a program.
- Operations on behaviors include:
  - Generic operations such as arithmetic, integration, differentiation, and time-transformation.
  - Domain-specific operations such as edge-detection and filtering for vision, scaling and rotation for animation and graphics, etc.

# Events in FRP

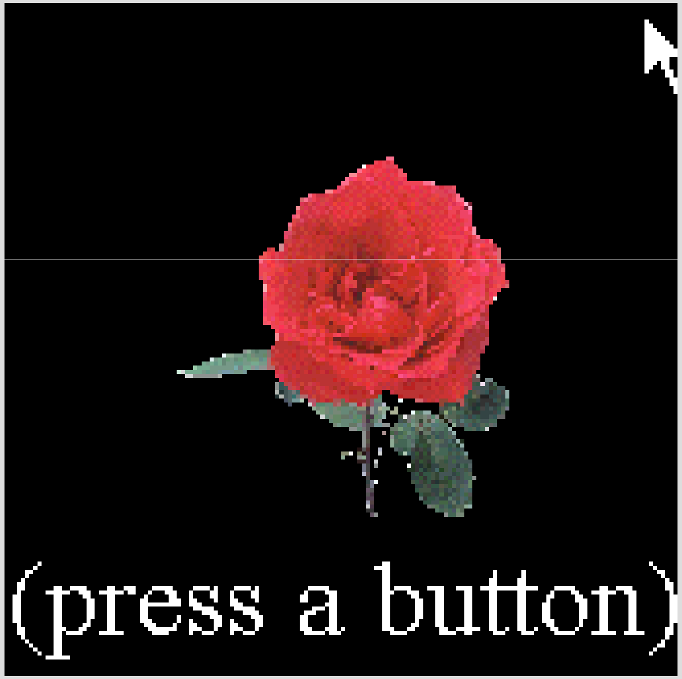
- Discrete event streams include user input as well as domain-specific sensors, asynchronous messages, interrupts, etc.
- They also include tests for dynamic constraints on behaviors (temperature too high, level too low, etc.)
- Operations on event streams include:
  - Mapping, filtering, reduction, etc.
  - Reactive behavior modification (next slide).

# An Example from Graphics (Fran)

A single animation example that demonstrates key aspects of FRP:

```
growFlower  =  stretch size flower
  where  size = 1 + integral bSign

bSign =
  0 `until`
    (lbp ==> -1 `until` lbr ==> bSign) .|.
    (rbp ==>  1 `until` rbr ==> bSign)
```



# Computer Music Apps
# Can Get Arbitrarily Complex

- We need the best languages, tools, programming environments, etc.
- (We also need the best algorithms, data structures, and so on.)
- Special-purpose computer-music languages have "issues":
  - often *too* special-purpose
  - sometimes marginal implementations
  - usually not designed by PL experts
  - huge overhead costs to implement and maintain

# Haskore and HasSound

- Domain-specific embedded languages for music and sound synthesis, respectively.
- Being “reborn” in the context of the Computing and the Arts initiative at Yale.
- Being used in two-course sequence in *Fundamentals of Computer Music:*
  - *Algorithmic and Heuristic Composition*
  - *Sound Representation and Synthesis*

# Functional Music Makes Sense

- Purely functional languages are especially suited to computer music.
- Declarative: saying "What" instead of "How“.
- Haskell's abstraction mechanisms allow musical programs that are elegant, concise, powerful:
  - higher-order functions
  - algebraic data types
  - lazy evaluation
  - type classes
- Aesthetics matter.

# Technology Has Improved

- Computers are *much* faster!!
- Implementations are *much* better!!
  - run faster
  - generate faster code
  - more user friendly
  - better programming environments
- Libraries are *much* more plentiful!!
- In particular, the GHC compiler, interpreter, and libraries are now “industrial strength.”

*“A large enough quantitative difference makes a qualitative difference.”*

# Design Goals for Haskore II

- The obvious:
  simplicity, expressiveness, generality, performance.
- Vertical design:
  - Good for signal processing / sound synthesis.
  - Good for algorithmic composition.
  - Good for reactive/interactive applications.
- Musical User Interface (MUI).
- Real-time sound synthesis.
- Seamless integration of the continuous and discrete.
- Transparency of design.

# Glove

composed and rendered
in Haskore by
Tom Makucivich
(a musician!)

with a little help from yours truly

# Haskore Basics

Simple representations of basic types:

```
type Octave = Int
type Dur = Rational
type Pitch = (PitchClass, Octave)
data PitchClass = Cff | Cf | C | Dff | Cs | Df | Css | D | Eff | Ds
        | Ef | Fff | Dss | E | Es | Ff | F | Gff | Ess | Fs | Gf | Fss | G
        | Aff | Gs | Af | Gss | A | Bff | As | Bf | Ass | B | Bs | Bss
data Prim a = Note Dur a | Rest Dur
```

For example:

```
Note (1/4) (C,4)  ::  Prim Pitch                -- Middle C quarter note
```

# The Music Type

```
data Music a = Primitive (Prim a)               – primitive note or rest
    | Music a :+: Music a                       -- sequential composition
    | Music a :=: Music a                       -- parallel composition
    | Modify Control (Music a)                  -- modifier

data Control =
      Tempo Rational                            -- scale the tempo
    | Transpose AbsPitch                        -- transposition
    | Instrument InstrumentName                 -- instrument label
    | Phrase [PhraseAttribute ]                 -- phrase attributes
    | Player PlayerName                         -- “player” label

type AbsPitch = Int                             -- absolute pitch
type PlayerName = String                        -- player names
type InstrumentName = AcousticGrandPiano
    | Vibraphone
    | Flute …                                   -- from General Midi standard
```

# For Convenience

- Constructor shorthands:

```
note d p         = Primitive (Note d p)
rest d           = Primitive (Rest d)
tempo r m        = Modify (Tempo r) m
transpose i m    = Modify (Transpose i) m
…
```

- Note and rest names:

```
c o d    = note d (C, o)
cs o d   = note d (Cs, o)
…
qn = 1/ 4;   qnr = rest qn
en = 1/ 8;   enr = rest en
…
```

- Example: ii-V-I chord progression in C major:

```
let dMin = d 3 qn :=: f 3 qn :=: a 3 qn
    gMaj = g 3 qn :=: b 3 qn :=: d 4 qn
    cMaj = c 3 hn :=: e 3 hn :=: g 3 hn
in dMin :+: gMaj :+: cMaj
```

# Higher-Order Functions

- How can any programmer (or artist!) do without them? ☺
- Two key data abstractions (as for lists): *map* and *fold*.
- First *map* (functor):

$$mMap :: (a \rightarrow b) \rightarrow Music\ a \rightarrow Music\ b$$

- Key property: $mMap\ id = id$
- For example:

**type** *Volume = Int*

$addVolume :: Volume \rightarrow Music\ Pitch \rightarrow Music\ (Pitch, Volume)$
$addVolume\ v = mMap\ (\lambda p \rightarrow (p, v))$

$scaleVolume :: Rational \rightarrow Music\ (Pitch, Volume) \rightarrow Music\ (Pitch, Volume)$
$scaleVolume\ r = mMap\ (\lambda(p,v) \rightarrow (p, round\ (r * v)))$

# Fold (catamorphism)

- More general than *mMap*.

  $mFold :: (b \to b \to b) \to (b \to b \to b) \to$
  $\quad (Prim\ a \to b) \to (Control \to b \to b) \to$
  $\quad Music\ a \to b$

- Key property:

  $mFold\ (:+:)\ (:=:)\ Primitive\ Modify = id$

- For example, to compute the *duration* of a *Music* value:

  *dur* :: *Music a* → *Dur*
  *dur* = *mFold* (+) *max getDur modDur* **where**
  *getDur* (*Note d p*) = *d*
  *getDur* (*Rest d*) = *d*
  *modDur* (*Tempo r*) *d* = *d* / *r*
  *modDur d* = *d*

# Super Retrograde

- Reverse (in time) an entire *Music* value.
- Requires a temporal semantics for *(:=:)*
- Recall: *flip f x y = f y x*

```
revM :: Music a → Music a
revM = mFold (flip (:+:)) (=:) Primitive Modify where
          m1 =: m2          = let d1 = dur m1
                                  d2 = dur m2
                              in if d1>d2 then m1 :=: (rest (d1-d2) :+: m2)
                                          else (rest (d2-d1) :+: m1) :=: m2
```

# Lazy Evaluation and Infinite Music

- It is perhaps not surprising that lazy evaluation is useful in many computer music apps.
- As a simple example:

  *repeatM* :: *Music a* → *Music a*
  *repeatM m* = *m* :+: *repeatM m*

- This motivates the need for a "truncating" parallel composition operator *(:=/)* such that *dur (m1 :=/ m2)* is equal to the *minimum* of *dur m1* and *dur m2*. Thus if one music value is infinite, it gets truncated to the length of the other one.

# Interpretation and Performance

- What does a Music value actually *mean?*

- An abstract *performance* is a sequence of musical *events:*

```
type Performance = [Event]
data Event       = Event PTime InstrumentName
                         AbsPitch DurT Volume
```

- The event *Event t i p d v* captures the fact that at time *t* instrument *i* sounds pitch *p* with volume *v* for a duration *d.*

# From Music to Performance

- To convert a *Music* value into a *Performance*, we need a *Context:*

**data** *Context a = Context*

| | |
|---|---|
| *Time* | -- time that music begins |
| *Player a* | -- default player |
| *InstrumentName* | -- default instrument |
| *DurT* | -- duration of one beat |
| *Key* | -- key (absolute pitch) |
| *Volume* | -- default volume |

- The function *perform* does the desired interpretation:

*perform :: Context a → Music a → Performance*

# Musical Equivalence

- Some *Music* values are not equal as Haskell values, but are equivalent musically, such as:

  *(m1 :+: m2) :+: m3* and *m1 :+: (m2 :+: m3)*

  (In other words, we expect *(:+:)* to be *associative.)*
- **Definition:** Two musical values *m1* and *m2* are *equivalent,* written *m1 ≡ m2,* if and only if:

  *(∀c) perform c m1 = perform c m2*

- In other words:

  *"if two things sound the same, they are the same"*
- The above equivalence can then be stated as an axiom:

  For all *m1*, *m2*, and *m3*:

  *(m1 :+: m2) :+: m3 ≡ m1 :+: (m2 :+: m3)*

# An Algebra of Music

- There are eight axioms that comprise an *algebra of music.*
- For example, *(:=:)* is not only associative, it is commutative.
- Another (important) example:

  **Duality of (:+:) and (:=:)**

  For any *m0, m1, m2,* and *m3* such that *dur m0 = dur m2:*

  *(m0 :+: m1) :=: (m2 :+: m3) ≡ (m0 :=: m2) :+: (m1 :=: m3)*
- Each axiom is provably sound.
- The axiom set is also *complete:* If two music values are equivalent, they can be proven so using only the eight axioms.
- Furthermore, the algebra can be made *polymorphic:* it is valid for video, audio, animation, even dance.
- *The Eight Laws of Polymorphic Temporal Media*.
- Allows designing languages having the same “look and feel” across a variety of base media types.


Available now at your neighborhood cafepress.com…

# Haskore's MUI
## (Musical User Interface)

- Design philosophy:
  - GUI's are important!
  - The dataflow metaphor ("wiring together components") is powerful!
  - Yet graphical *programming* is inefficient…
- Goal: *an effective set of UI widgets that can be programmed using a dataflow metaphor at the linguistic level*.
- We achieve this via two levels of abstraction:
  - The *UI Level*
    - Create widgets
    - Attach titles, labels, etc.
    - Control layout
  - The *Signal Level*
    - A signal is conceptually a *continuous* *(time-varying) value*.
    - Sliders, knobs, etc. provide are input widgets.
    - Midi out, graphics, etc. are output widgets.

# Signals

- Signals are *time-varying quantities*.
- Conceptually they can be thought of as functions of time:

  *Signal a* = Time → a

- For example, the output of a slider is a time-varying *number*.
- Key idea: Lift all static functions to the signal level using a family of *lifting functions*:

  *lift0 :: a → Signal a*
  *lift1 :: (a → b) → (Signal a → Signal b)*
  *lift2 :: (a → b → c) → (Signal a → Signal b → Signal c)*
  *…*

- Haskell's type classes make this especially easy.
- Conceptually:

  $s_1 + s_2 \quad = \quad \lambda t \rightarrow s_1\, t + s_2\, t$
  $\sin s \quad = \quad \lambda t \rightarrow \sin (s\, t)$
  *…*

- One can also *integrate* signals.

# Events

- Signals are not enough… some things happen *discretely*.
- *Events* can be realized as a kind of signal:

  **data** *Maybe a = Nothing | Just a*
  **type** *EventS a = Signal (Maybe a)*

- So events are actually *event streams*.
- Midi event streams simply have type:

  *EventS [MidiMessage]*

  where *MidiMessage* encodes standard Midi messages such as Note-On, Note-Off, etc.
- In addition:

  *midiIn :: Signal DeviceID → UI (EventS [MidiMessage])*
  *midiOut :: Signal DeviceID → EventS [MidiMessage] → UI ()*

# MUI Examples

- Pitch translator:

```
do ap ← title "Absolute Pitch" (hiSlider 1 (0, 100) 0)
   title "Pitch" (display (lift1 (show ◦ pitch) ap))
```

- Output Midi note when pitch changes:

```
do ap ← title "Absolute Pitch" (hiSlider 1 (0, 100) 0)
   title "Pitch" (display (lift1 (show ◦ pitch) ap))
   let ns = unique ap =>> (λk → [ANote 0 k 100 0.1])
   midiOut 0 ns
```

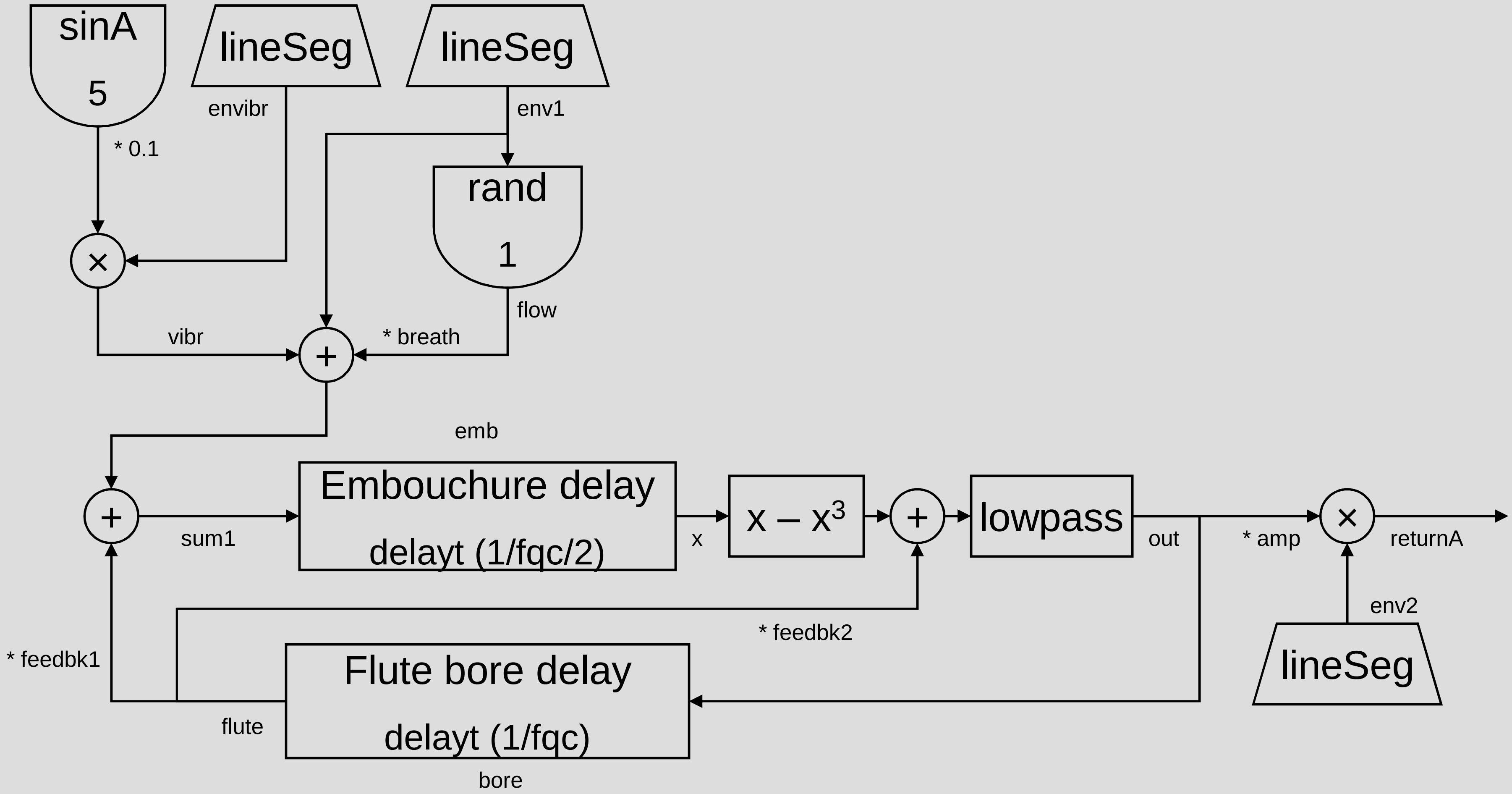
- Output Midi note at constant rate:

```
do …
   t ← time
   f ← title "Tempo" (hSlider (1, 10) 1)
   let ticks = timer t (1/f )
   let ns = snapshot ticks ap =>>(λk → [ANote 0 k 100 0.1])
   midiOut 0 ns
```

# HasSound

- HasSound is the piece of Haskore that focuses on signal/audio processing and sound synthesis.
- It uses a more sophisticated notion of *signal*, but is conceptually very similar.
- Supports *multiple clock rates* using phantom types and type classes.
- The correspondence between the mathematics and the program is very strong:  even recursive signals work.
- We can generate real-time sound at 44.1 KHz for moderately-sized instruments.  This will get better through optimization.

# Physical Model of a Flute



# Expressed in HasSound

```
flute :: Double -> AR -> Double -> CR -> AR -> AR
flute dur amp fqc press breath =
  let kenv1    = lineSeg [0, 1.1, 1, 1, 0] [0.06, 0.2, dur-0.16, 0.02]  :: CR
      kenv2    = lineSeg [0, 1, 1, 0] [0.01, dur-0.02, 0.01] :: CR
      kenvibr  = lineSeg [0, 0, 1, 1] [0.5, 0.5, dur-1] :: CR
      bore     = delayt (1/fqc)
      emb      = delayt (1/fqc/2)
      feedbk1  = 0.4
      feedbk2  = 0.4
      env1     = upSample (kenv1 * press)
      env2     = upSample kenv2
      envibr   = upSample kenvibr
      flow     = rand 1 env1
      vibr     = sinA 5 * 0.1 * envibr
      sum1     = breath * flow + env1 + vibr
      flute    = bore out
      x        = emb (sum1 + flute * feedbk1)
      out      = lowpass 0.27 (x - x**3 + flute * feedbk2)
  in out * amp * env2
```

# Sample Results

- f0 and f1 demonstrate the change in the breath parameter.

  f0 = flute 3 0.35 440 0.93 0.02
  f1 = flute 3 0.35 440 0.93 0.05

- f2 has a weak pressure input so it only plays the blowing noise.

  f2 = flute 3 0.35 440 0.53 0.04

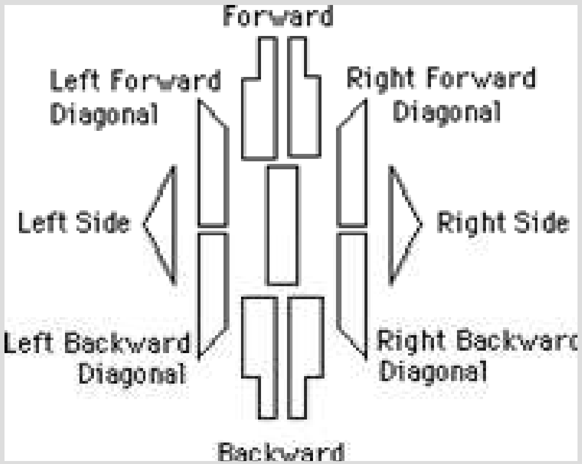
- f3 takes in a gradually increasing pressure signal.

  f3 = flute 4 0.35 440 (lineSeg [0.53, 0.93, 0.93] [2, 2]) 0.03

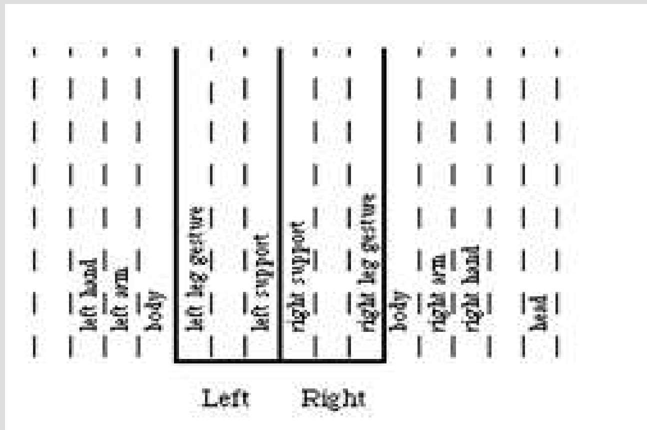
- Sequence of notes

# Dance!

- *Labanotation* is a notation for recording any kind of human movement.
- Introduced by (Austrian-) Hungarian Rudolf von Laban (1879-1958) in 1928.  In the US development continued, most notably Ann Hutchinson Guest.
- Shapes represent movement:

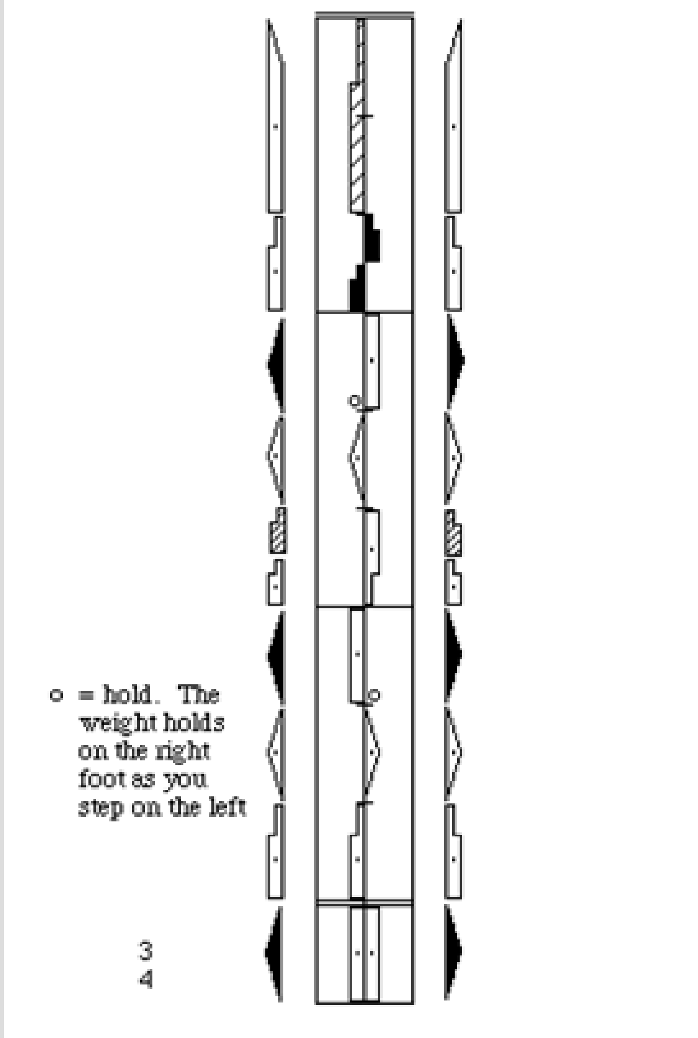



Columns represent body parts:



# Dance, cont'd

- Labanotation can be captured in an algebraic datatype not unlike Haskore.
- Was used to control humanoid robots in Liwen Huang's PhD thesis [Liwen2007].
- Can it instead be used to create languages and tools to help animators, dancers, actors, choreo-graphers, and playwrites?



# Conclusions

- "Computational Thinking" is finding its way into many disciplines, including the arts.
- Not just for traditional art – new modes are emerging, including interactive / dynamic art.
- Providing our finest ideas, languages, and tools is a good way for Computer Science to have an impact.
- Haskell and functional programming in general are potentially a good match for this "new way of thinking."

# Thank You!!

## (any questions?)