Deckbuild: A Declarative Domain-Specific Language for Card Game Design

Karl Cronburg, Raoul Veroy, Matthew Ahrens

Tufts University

Background - DeckBuild Language

- Domain - deck-based card games
- Users - card game designers & programmers
- Goals
  - Readability
  - User productivity
  - Multiple artifacts
- Existing languages
  - Primarily markup languages for card description
  - General purpose languages for describing rules of a game
  - Very ad-hoc - users need a domain-specific way to abstract away implementation specifics of general purpose languages

Language Features

- Can declaratively enumerate
  - Cards - name, type, effects, cost
  - Rulesets - e.g. how many cards you draw per turn
- Can define complex card effects directly from Haskell

Deck-Building Card Game Grammar

Raoul Veroy, Matthew Ahrens, Karl Cronburg

October 14, 2014

⟨*deckDecls*⟩ ::= ⟨*deckDecl*⟩*

⟨*deckDecl*⟩ ::= ⟨*cardDecl*⟩ | ⟨*turnDecl*⟩

⟨*cardDecl*⟩ ::= `card` ⟨*cardID*⟩ `::` ⟨*cardType*⟩ `{` ⟨*cardDescr*⟩ `}` `costs` ⟨*intLit*⟩

⟨*cardDescr*⟩ ::= ⟨*effectDescr*⟩* ⟨*englishDescr*⟩

⟨*englishDescr*⟩ ::= ⟨*stringLit*⟩*

⟨*effectDescr*⟩ ::= ⟨*plusOrMinus*⟩⟨*intLit*⟩ ⟨*effectType*⟩

⟨*plusOrMinus*⟩ ::= `+` | `-`

⟨*turnDecl*⟩ ::= `turn` ⟨*turnID*⟩ `{` ⟨*phaseDescr*⟩* `}`

⟨*phaseDescr*⟩ ::= ⟨*phaseName*⟩ ⟨*phaseInt*⟩

⟨*phaseInt*⟩ ::= ⟨*intLit*⟩ | `all`

⟨*effectType*⟩ ::= `actions` | `coins` | `buys` | `cards` | `victory`

⟨*cardType*⟩ ::= `Treasure` | `Action` | `Victory`

⟨*phaseName*⟩ ::= `action` | `buy` | `discard` | `draw`

⟨**ID*⟩ ::= ⟨*identifier*⟩

⟨**Lit*⟩ ::= As defined in Text.Parsec.Token

⟨*identifier*⟩ ::= As defined in Text.Parsec.Token

Example Program - Input

```
[deck|
card Cellar  :: Action {
  +1 actions
  "Discard any number of cards."
  " +1 Card per card discarded"
} costs 2

card Chapel  :: Action {
  "Trash up to 4 cards from your hand"
} costs 2

card Village :: Action {
  +1 cards
  +2 actions
} costs 3

card Woodcutter :: Action {
  +1 buys
  +2 coins
} costs 3

card Copper :: Treasure {
  +1 coins
} costs 0

card Silver :: Treasure {
  +2 coins
} costs 3

card Gold :: Treasure {
  +3 coins
} costs 6

card Estate :: Victory {
  +1 victory
} costs 2

card Duchy :: Victory {
  +3 victory
} costs 5

card Province :: Victory {
  +6 victory
} costs 8

turn Dominion_Standard {
  action 1
  buy 1
  discard all
  draw 5
}

|]
```

Example Program - Quasiquoter & Code Generator

```
things_derived = [mkName "Eq", mkName "Typeable", mkName "Show", mkName "Ord"]

make_deck_declaration :: [DeckDecl] -> Q [Dec]
make_deck_declaration ds = do
    card_es <- mapM genDeckExp $ filter      isCard  ds :: Q [Exp]
    turn_es <- mapM genDeckExp $ filter (not.isCard) ds
    let kingdomCardBody = NormalB $ ListE card_es
    let turnRulesBody   = NormalB $ ListE turn_es
    let name_constructors = genCons ds
    let card_constructors = genCardCons
    return [ DataD [] (mkName "CardName")     [] name_constructors things_derived
           , DataD [] (mkName "RuntimeCard")  [] card_constructors things_derived
           , FunD     (mkName "kingdomCards") [Clause [] kingdomCardBody []]
           , FunD     (mkName "turnRules")    [Clause [] turnRulesBody   []]
           ]

genCons :: [DeckDecl] -> [Con]
genCons ds = [NormalC (mkCardName d) [] | d <- ds, isCard d]

genCardCons :: [Con]
genCardCons = [ RecC (mkName "RuntimeCard")
                [ (mkName "cID",    IsStrict, ConT $ mkName "CardName")
                , (mkName "cType",  IsStrict, ConT $ mkName "CardType")
                , (mkName "cDescr", IsStrict, ConT $ mkName "CardDescr")
                , (mkName "cCost",  IsStrict, ConT $ mkName "CardCost")
                ]
              ]

isCard :: DeckDecl -> Bool
isCard (DeckDeclCard c) = True
isCard _                = False

mkCardName :: DeckDecl -> Name
mkCardName (DeckDeclCard (Card {cID = name})) = mkName $ map toUpper name
mkCardName _ = undefined

genDeckExp :: DeckDecl -> Q Exp
genDeckExp e = liftD e --runQ [| e |]
```

Example Program - CodeGen Output

```
data CardName = CELLAR | CHAPEL | VILLAGE | WOODCUTTER | COPPER
              | SILVER | GOLD   | ESTATE  | DUCHY      | PROVINCE
deriving (Eq, Typeable, Show, Ord)

data RuntimeCard = RuntimeCard { cID    :: !CardName
                               , cType  :: !CardType
                               , cDescr :: !CardDescr
                               , cCost  :: !CardCost }
deriving (Eq, Typeable, Show, Ord)

kingdomCards = [ RuntimeCard
                   { cID    = CELLAR
                   , cType  = ACTION
                   , cDescr = CardDescr
                               { primary = [Effect {amount = 1, effectType = ACTIONS}]
                               , other   = "Discard any number of cards. +1 Card per card discarded"
                               }
                   , cCost   = 2 }

                 , ...

                 , RuntimeCard
                   { cID    = PROVINCE
                   , cType  = VICTORY
                   , cDescr = CardDescr
                               { primary = [Effect {amount = +6, effectType = VICTORYPOINTS}]
                               , other   = []
                               }
                   , cCost = 8 }
                ]

turnRules = [ Turn
              { turnID     = "Dominion_Standard"
              , turnPhases =
                [ Phase {phaseName = ActionP,  phaseInt = PhaseInt 1 }
                , Phase {phaseName = BuyP,     phaseInt = PhaseInt 1 }
                , Phase {phaseName = DiscardP, phaseInt = All        }
                , Phase {phaseName = DrawP,    phaseInt = PhaseInt 5 }
                ] } ]
```

Example Program - Complex Card Effects

- Can reference quasiquoted EDSL code from Haskell
- Future: design imperative-friendly (non-Haskell) EDSL for specifying complex card effects
- Example below - implementation of `CELLAR` effects


```haskell
-- Discards any number of cards, returning the number of cards discarded
cellarEffect' :: forall (m :: * -> *). (MonadIO m, MonadState Game m) => m Int
cellarEffect' = do
  g  <- get
  c' <- liftIO $ ((mayPick.p1) g) g CELLAR
  case c' of
    Just c  -> if   elem c ((cards.hand.p1) g)
               then discard c >> cellarEffect' >>= \n -> return $ n + 1
               else return 0
    Nothing -> return 0

-- Discard any number of cards, then draw that many cards:
cellarEffect :: forall (m :: * -> *). (MonadIO m, MonadState Game m) => m ()
cellarEffect = addActions 1 >> cellarEffect' >>= \n -> draw n
```

[1] card image & content by D. X. Vaccarino

Demonstration

Run-time System

```
> runGreedy (0.5, 0.5)
Player1:
    name   = "Greedy1"
    hand   = [ESTATE, GOLD, PROVINCE, PROVINCE, SILVER]
    inPlay = []
    deck   = [SILVER,   PROVINCE, COPPER, SILVER,  COPPER
             ,ESTATE,   COPPER,   COPPER, VILLAGE, VILLAGE
             ,PROVINCE, ESTATE,   SILVER, COPPER]
    dscrd  = [SILVER, SILVER, SILVER, COPPER, COPPER, PROVINCE]
    buys=1, actions=1, money=0

Player2:
    name   = "Greedy2"
    hand   = [COPPER, COPPER, COPPER, COPPER, VILLAGE]
    inPlay = []
    deck   = [SILVER,   SILVER, GOLD, COPPER, COPPER,   ESTATE
             ,GOLD,     ESTATE, GOLD, ESTATE, PROVINCE, VILLAGE
             ,PROVINCE, GOLD]
    dscrd  = [SILVER, COPPER, SILVER, SILVER, SILVER, PROVINCE]
    buys=1, actions=1, money=0

Trash: []
Supply: [(COPPER,60),  (CELLAR,10),  (MOAT,10),       (ESTATE,8)
        ,(SILVER,27),  (VILLAGE,6),  (WOODCUTTER,10), (WORKSHOP,10)
        ,(MILITIA,10), (REMODEL,10), (SMITHY,10),     (MARKET,10)
        ,(MINE,10),    (DUCHY,8),    (GOLD,25),       (PROVINCE,0)]
Turn #: 30
```

Evaluation

- Ease of use to describe all existing dominion cards
- Time to program the equivalent cards in other data Languages: XML, JSON, YAML
- In next iteration, when able to express card effects
  - How english-like expressing difficult card effects is
  - How complex the card effects can be
  - How intuitive it is to do both at the same time

Future - Tool Support

- Type-checker
- Debugging information
- Automated card *balancer* (e.g. 'Maybe card XXX should be $1 cheaper.')
- Develop more complex card examples for reference use
- Library of AI players using various heuristics
- Markup language conversion tools / support
- Graphical client-server support

THE MOST POWERFUL GAMING SYSTEMS IN THE WORLD STILL CAN'T MATCH THE ADDICTIVENESS OF TINY IN-BROWSER FLASH GAMES.

[1] image from xkcd.com/484

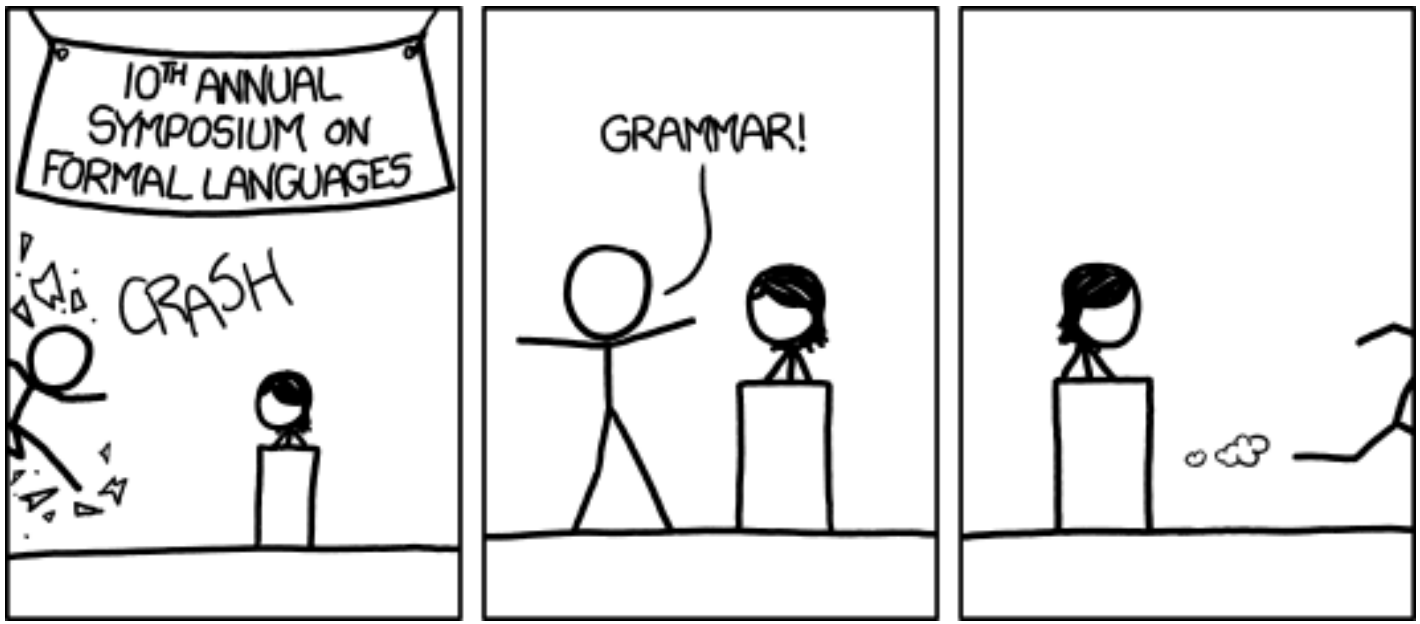
Future - Language Goals

- Language support for English-like effect descriptions
- Card-balancing algorithm
- Comma-delimited syntactic sugar

Questions?

Acknowledgements:

- Kathleen for Haskell and Language expertise
- learnyouahaskell.com



[audience looks around] 'What just happened?' 'There must be some context we're missing.'

[1] image from xkcd.com/1090