30% Demonstration

Andrew Hirsch

The George Washington University

November 28, 2012

The Form of Data

- Datatypes: "Sum" of constructors
- Constructors: functions F that take data of type a into data Fa

```
data Directions = North Int
                | South Int
                | East Int
                | West Int
```

```
data Directions' = North' :+: South' :+: East' :+: West'
data North' = North' Int
data South' = South' Int
data East'  = East' Int
data West'  = West' Int
```

$$\texttt{Directions} \cong \texttt{Directions}'$$

The Form of Data, Ctd.

- What about recursive data types?

```
data ListInt = Nil
             | Cons Int ListInt

data ListInt' = Nil :+: Cons
data Nil e = Nil
data Cons e = Cons Int e
```

- What's special about Cons?
  - Cons is a *functor*
  - Cons should have a function `fmap :: (a -> b) -> (Cons a -> Cons b)`
  - Nil is also a functor, trivially
  - ListInt' is also a functor

$$\texttt{ListInt} \cong \texttt{ListInt}'?$$

Tying the Recursive knot

```haskell
data Term f = f (Term f)
```

- $\texttt{ListInt} \cong \texttt{Term}(\texttt{ListInt}')$
- "Tying the Recursive Knot"

Parsing into Tied Structures

```
parseListInt :: Parser ListInt'
parseListInt = parseCons <|> parseNil

parseCons :: Parser ListInt'
parseCons = do
  i <- parseInt
  char ':'
  l <- parseListInt
  return $ iCons i l

parseNil :: Parser ListInt'
parseNil = do
  string "[]"
  return iNil
```

Parsing into Untied Structures

```haskell
parseListInt :: Parser e -> Parser (ListInt' e)
parseListInt p = (do
                    c <- parseCons e
                    return $ inr c)
             <|> (do
                    n <- parseNil e
                    return $ inl c)

parseCons :: Parser e -> Parser (Cons e)
parseCons p = do
  i <- parseInt
  char ':'
  e <- p
  return $ Cons i p

parseNil :: Parser e -> Parser (Nil e)
parseNil p = do
  string "[]"
  return Nil
```

Tying while parsing?

- Can we use the code last slide to get a parser for `Term(ListInt')`?
- Need some sort of *fixed point* for parsers
- None currently exists

Solution: Side Step

- We instead introduce grammars with inheritance
- Grammars can *extend* other grammars
- Those then get translated into a full grammar
- Grammars have ADTs and Happy files generated automatically
- Happy = YACC for Haskell

Example: EBNF for EBNF

```
Grammar EBNF {

EBNF ::= {Production}

Production ::= Identifier "::=" Expression ".".

Expression ::= Term {"|" Term}.

Term ::= Factor {Factor}.

Factor ::= Identifier
         | "[" Expression "]"
         | "(" Expression ")"
         | "{" Expression "}"
         | Literal.

Identifier ::= Character { Character }.

Literal ::= "'" Character { Character }
          | '"' Character { Character }.

}
```

Example: EBNF Subgrammar for EBNF Subgrammars

```
Grammar EBNFSubgrammar extends EBNF {

Production ::+ Identifier "::+" Expression.

}
```