EBNFRepr-0.1

Xavier Ho, s2674674

May 9, 2011

Abstract

EBNFRepr visualises Extended Backus-Naur Form (EBNF) in Encapsulated PostScript format (EPS) and Scalable Vector Graphics format (SVG), written in Haskell.

1 Using EBNFRepr

EBNFRepr-0.1 is written against the following dependencies:

Wumpus-Core-0.43.0 MissingH-1.1.0

Note: Wumpus-Core.0.50.0 broke backwards compatibility, and will not work with EBNFRepr-0.1. Please make sure your hackage list has the correct version.

1.1 Building from source code

The distribution comes with source code written in Haskell, written against the Glasgow Haskell Compiler (GHC). Once your system has GHC installed, and its cabal up-to-date, you can type the command:

```
> ghc --make Main
```

This will create the executable binary for EBNFRepr. Move it to a directory of your preference. If you are on a Windows system, alternatively you can run the batch file to compile:

```
> run
```

This will compile and parse the basic EBNF files supplied in the syntax folder, and generate them into the output folder.

1.2 Running EBNFRepr

Once the executable is created, you can run the command:

```
> Main syntax/production.ebnf
```

This will parse the EBNF grammar[1] (and its embedded metadata) and generate the corresponding visual format in both EPS and SVG, in the output folder. To use a different file, simply change the path in the command line argument.

1.3 Viewing the results

Most browsers today support the SVG format. To view a generated picture, simply open up the .SVG file in a browser. The EPS format can be converted to PDF format using tools like *epstopdf*, and viewed in Adobe Reader.

The next chapter highlights some of the core features of EBNFRepr. For function-level documentation, the source code itself provides additional comments.

2 High-level overview

On a very high-level, EBNFRepr's architecture is as follows[2]:

```
+------------------+       +-----------------+
| EBNF Definitions |       | Visualised EBNF |
+------------------+       +-----------------+
    v Input v                       |
+--------------------+- Produces ---+
| EBNFRepr (Main.hs) |------ Uses --+
+--------------------+              |
      v Uses v                      |
  +---------------+                 |
  | EBNFParser.hs |                 |
  +---------------+                 |
      v Uses v                      |
   +-------------+                  |
   | EBNFRepr.hs |                  |
   +-------------+                  |
      v Uses v                      v
     +--------+                 +--------+
     | Geo.hs |----- Uses ----- > | Wumpus |
     +--------+                 +--------+
```

Wumpus is a SVG/EPS drawing library written in Haskell. **Geo.hs** contains functions to draw basic primitives, such as a filled rectangle, or a text label. **EBNFRepr.hs** further uses these functions to create EBNF-specific pictures.

EBNFParser.hs has very little drawing concern. Its function is to read a string of EBNF definitions, and parses it into a meaningful data structure for drawing. **Main.hs** is a simple program that reads from a file, and delegates the drawing into the previous modules.

[1]The supplied grammar is an extended EBNF format created by Andrew Rock, http://www.ict.griffith.edu.au/arock/

[2]Unfortunately, neither EPS or SVG is supported by pdflatex, and so ASCII art comes to the rescue.

3 Geo module

The Geo module supplies both the definitions of a few primitives such as rectangles and text labels, as well as the functions for drawing them. The data type

```
type Component = (DPicture, DVec2)
```

is the key player that allows the drawing to cascade and/or chain after one another. A Component is made up of two things: a picture and an end point. Its start point is implied at $(0, 0)$, but in relative coordinates. Chaining up two or more Components allow the coordinates to accumulate, and shifted to the right places in absolute coordinates upon drawing.

There are two ways to draw in the module Geo; draw, and drawBranch:

```
draw :: [Component] -> Component
draw components = draw' components identityMatrix [] (V2 0 0)

drawBranch :: [Component] -> Component
drawBranch components =
  (fst $ draw' components identityMatrix [] (V2 0 0), (V2 0 0))

draw' :: [Component] -> DMatrix3'3 -> [DPicture] -> DVec2
                                                        -> Component
draw' [] _ pictures point = (multi pictures, point)
draw' (component:components) matrix pictures (V2 x y) =
  let shifted_picture = transform matrix $ fst component
      tx = vector_x $ snd component
      ty = vector_y $ snd component
      new_matrix = matrix * translationMatrix tx ty
  in draw' components new_matrix
       (pictures ++ [shifted_picture]) (V2 (x+tx) (y+ty))
```

The difference between the two functions is very straight forward: **draw** allows the end point to be set at the last Component's end point; **drawBranch** ignores that and sets the end poitn to be the same as the start point. This allows branching in drawing diagrams.

The other important feature of note is that both draw and drawBranch merges a list of Components into one Component. This allows a draw function to be nested inside itself, or inside a drawBranch function, and vice versa. It makes use of the powerful concept that is recursion.

Inside Geo, there are mainly two types of drawing functions: Component-level drawing functions and Absolute drawing functions.

Component-Level drawing functions all return the type of Component. They allow drawing in relative coordinates, and supplies the appropriate end points for each individual picture.

Absolute drawing functions are very tightly integrated with Wumpus. They return the type of DPicture, allowing pixel-perfect drawing in absolute coordinates. This is rarely used in EBNFRepr.hs; however, the dashed box drawing is achieved with absolute coordinates simply because it was easier that way.

4 EBNFRepr module

In EBNFRepr module, more Component-level drawing functions are presented, along with several shorthand functions and constants. They utilise the low-level functions in Geo module, supplying the simplest interface possible for ease of use.

```
textLabel :: String -> Component
textLabel = text black

drawTerminal :: String -> Component
drawTerminal = roundTextBox black green black

drawNonterminal :: String -> Component
drawNonterminal = textBox black orange black

drawSpecial :: String -> Component
drawSpecial s
  | s == "epsilon"  = epsilon
  | s == "..."      = textLabel s
  | otherwise       = textBox black yellow black s
```

There is a one-to-one mapping between the functions and the EBNF definitions. A **textLabel** is a simple textbox without any borders or background fill. **drawTerminal** draws a rounded textbox with green background. **drawNonterminal** draws a rectangle textbox with an orange background. drawSpecial draws, most of the time, a yellow rectangle textbox. It may draw nothing, or a textLabel of "...", when appropriate.

As mentioned previously, all of the Component-level functions are in relative coordinates. You can see they take no start or end coordinates; the start is assumed to be the previous point—first one being $(0, 0)$—and the end point is relative and specific to each type of pictures. The size of the picture can be calculated with its contents, often the textLabel itself, or another drawing entirely.

Our EBNF visualisation uses the railroad diagram. The following functions allow drawing them easily:

```
hRail :: Double -> Component
hRail = hLine black

vRail :: Double -> Component
vRail = vLine black

rdRail = rightRoundDown black
drRail = downRoundRight black
ruRail = rightRoundUp black
ulRail = upRoundLeft black
ldRail = leftRoundDown black
urRail = upRoundRight black

epsilon :: Component
epsilon = hRail 0
```

hRail and **vRail** are horizontal and vertical lines, with an input of its length. The next 6 funtions are for railroad turns; rdRail means *going towards the right and turning downwards*, etc. This is intuitive once you start using it, trust me. Finally, **epsilon** is effective an empty drawing, implemented with a zero-length rail.

The rest of the functions in EBNFRepr.hs are higher-level representation drawing functions.

```
diagram :: String -> [Component] -> [Component]
diagram s components =
  [textLabel s]
  ++ [hRail default_font_size_px]
  ++ (terminals components)
  ++ [hRail default_font_size_px]

drawDiagram :: String -> [Component] -> Component
drawDiagram s = (draw . diagram s)

branchDiagram :: String -> [Component] -> Component
branchDiagram s = (drawBranch . diagram s)
```

For instance, **diagram** takes a list of components, and prepends and postpends two horizontal rails front to back, along with a label on top. All of the higher-level drawing functions return a list of Components.

For each higher-level drawing function, two types of drawing functions are supplied to simplify the often usage between draw and drawBranch. Depending on the type of the drawing, they could return Component, or a list of Components, as we will see in **terminals**:

```
terminals :: [Component] -> [Component]
terminals (component:components) =
  terminals' components [component]
    where
    terminals' [] connected = connected
    terminals' (component:components) connected
      = terminals' components
      (connected ++ [hRail (default_font_size_px*8/5), component])

drawTerminals :: [Component] -> [Component]
drawTerminals c = [draw $ terminals c]

branchTerminals :: [Component] -> [Component]
branchTerminals c = [drawBranch $ terminals c]
```

terminals simply connect each Component with a rail inbetween. Again, two more drawing functions are supplied for it, one for continuous drawing and one for branched drawing. This time, they return the type of a list of Components, because it was intuitive to design them that way.

The following functions are all defined in EBNFRepr. Their shorthand functions for continuous drawing and branched drawing have been omitted for space.

```
oneOrMany :: [Component] -> [Component]
oneOrMany components =
  [ inner
  , ldRail
  , vRail (-height)
  , drRail
  , hRail width
  , ruRail
  , vRail height
  , ulRail
  ] where
    inner = drawBranch components
    width = boundaryWidth $ boundary $ fst inner
    height = (boundaryHeight $ boundary $ fst inner)
             - default_font_size_px / 2

optional :: [Component] -> [Component]
optional components =
  [ optionalComponents
  , hRail outer_width
  ] where
    inner = draw components
    optionalComponents = drawBranch
      [ rdRail
      , drRail
      , inner
      , ruRail
      , urRail
      ]
    width = boundaryWidth $ boundary $ fst inner
    outer_width = boundaryWidth $ boundary $ fst optionalComponents
    height = (boundaryHeight $ boundary $ fst inner)
             - default_font_size_px / 2

zeroOrMany :: [Component] -> [Component]
zeroOrMany components =
  [ drawBranch [hRail width]
  , ldRail
  , drRail
  , inner
  , ruRail
  , ulRail
  ] where
    inner = draw components
    width = (boundaryWidth $ boundary $ fst inner)
    height = (boundaryHeight $ boundary $ fst inner)
             - default_font_size_px / 2

alternative :: [Component] -> [Component]
alternative (component:components) =
  [ drawBranch [ hRail default_font_size_px
               , component
               , hRail trailingWidth]
  , rdRail
  , draw $ alternative' components maxWidth height
  , urRail
  ] where
    maxWidth = maximum
      $ map (boundaryWidth . boundary . fst) (component:components)
    trailingWidth = maxWidth - width + default_font_size_px
    width = boundaryWidth $ boundary $ fst component
    height = boundaryHeight $ boundary $ fst component

alternative' [] _ _ = [epsilon]
alternative' (component:components) maxWidth offset =
  [ vRail (-offset)
  , drawBranch $ alternative' components maxWidth offset'
  , drRail
  , component
  , hRail trailingWidth
  , ruRail
  , vRail offset
  ] where
    trailingWidth = maxWidth - width
    width = boundaryWidth $ boundary $ fst component
    height = boundaryHeight $ boundary $ fst component
    offset' = (max height default_font_size_px)
            + default_font_size_px/2

except :: [Component] -> String -> [Component] -> [Component]
except a s b =
  dashedBox contents
    where
    maxWidth = max aWidth bWidth
    aWidth = boundaryWidth $ boundary $ fst $ drawBranch a
    bWidth = boundaryWidth $ boundary $ fst $ drawBranch b
    aTrailing = maxWidth - aWidth
    bTrailing = maxWidth - bWidth
    a' = [epsilon] ++ a ++ [epsilon]
    b' = [epsilon] ++ b ++ [epsilon]
    contents =
      [ drawBranch [ textLabel ""
                   , textLabel s
                   , draw $ terminals b'
                   , hRail bTrailing]
      , draw $ terminals a'
      , hRail aTrailing
      ]

dashedBox :: [Component] -> [Component]
dashedBox contents =
  [ drawBranch
    [(drawDashedRect black
        x (y+default_font_size_px * 5 / 8)
        width (-(height-default_font_size_px/4)),
    V2 x y)]
  , inner
  ] where
    x = 0
    y = 0
    width = boundaryWidth $ boundary $ fst inner
    height = boundaryHeight $ boundary $ fst inner
    inner = draw contents
```

5 EBNFParse module

This module heavily borrows from Andrew Rock's EBNF Grammar package, found on his personal homepage. In particular, the Syntrax[3] package. We shall go through the data structures quickly.

```
type Nonterminal = String
type Terminal = String
type Special = String
```

There are three types of text in this particular flavour of EBNF. A Terminal is a string literal encapsulated in double quotes ("). A Nonterminal is an identifier, made up of letters(a-z,A-Z), digits (0-9), underscores (_), and apostrophes ('). A Special is a string literal encapsulated in dollar ($) signs.

```
infix 5 :&, :!
data Expression = Terminal Terminal
                | Nonterminal Nonterminal
                | Special Special
                | OR [Expression]
                | Many Expression
                | Some Expression
                | Optional Expression
                | Seq [[Expression]]
                | Expression :& Expression
                | Expression :! Expression
  deriving (Show)
```

A Expression can be one of the following: a Terminal, a Nonterminal, a Special, a list of Expressions, one or more Expressions (Many), zero or more Expressions (Some), zero or one Expression (Optional), a sequence of Expressions (Seq, each inner list representing a line), two Expressions together with an ALSO or a NOT tag.

```
data Production = Production Nonterminal Expression [(Nonterminal, Terminal)]
  deriving (Show)
```

[3]The source code of Syntrax can be found at http://www.ict.griffith.edu.au/arock/haskell/index.html.

A Production has a Nonterminal (used to title), followed by an Expression, and some metadata in Nonterminal, Terminal pairs. The Expression and the metadata are separated by semicolons (;), and so does in between each metadata. The end of a Production is noted by a full stop (.).

```
data Grammar = Grammar [Production]
  deriving (Show)
```

A Grammar is a list of Productions, allowing multiple productions to be specified in one file.

5.1 Parsing

EBNFRepr uses Text.Parsec, the standard parser distributed with Haskell. We define a simple language definition used to create the parser:

```
styleDef :: LanguageDef st
styleDef = emptyDef
         { commentLine = "#"
         , nestedComments = False
         , identStart = letter <|> digit <|> (oneOf "_") <|> (oneOf "’")
         , identLetter = alphaNum <|> oneOf "_"
         , opStart = oneOf "|({[+.:\\<&!;"
         , opLetter = oneOf ")}]:="
         , caseSensitive = True
         }

lexer :: TokenParser ()
lexer = makeTokenParser styleDef
```

Once we have the lexer, we can create parsers based on its style. These parsers conform to each definition within the EBNF, also found on Syntrax's documentation.

```
production :: Parser Production
production = do
  name <- identifier lexer
  symbol lexer "::="
  expr <- expression
  metas <- many metadata
  symbol lexer "."
  return $ Production name expr metas

metadata :: Parser (Nonterminal, Terminal)
metadata = do
  symbol lexer ";"
  n <- identifier lexer
  symbol lexer "="
  t <- stringLiteral lexer
  return (n, t)
```

```
expression :: Parser Expression
expression = do
  a <- alternate
  e <- expression'
  if length e == 0 then
    return a
  else
    return $ OR $ a:e

expression' :: Parser [Expression]
expression' =
      try (do symbol lexer "|"
              a <- alternate
              e <- expression'
              return $ a:e)
  <|> return []

alternate :: Parser Expression
alternate = do
  t <- terms
  a <- alternate'
  if length t > 1 then
    if length a > 0 then
      return $ Seq $ t:a
    else
      return $ Seq [t]
  else
    return $ head t

alternate' :: Parser [[Expression]]
alternate' =
  (do t <- many alternate''
      return t)
  <|> return []

alternate'' :: Parser [Expression]
alternate'' = do
  symbol lexer "\\"
  t <- terms
  return t

terms :: Parser [Expression]
terms = many1 term

term :: Parser Expression
term =
      try (do s <- terminal
              return s)
  <|> try (do s <- nonterminal
              return s)
  <|> try (do symbol lexer "$"
              s <- many1 (noneOf "$")
              symbol lexer "$"
              return $ Special $ replace "\\\\n" "\\n" s)
  <|> try (do symbol lexer "("
              s <- expression
              symbol lexer ")"
              return $ s)
  <|> try (do symbol lexer "["
              s <- expression
              symbol lexer "]"
              return $ Optional s)
  <|> try (do symbol lexer "{"
              s <- expression
              symbol lexer "}+"
              return $ Many s)
  <|> try (do symbol lexer "{"
              s <- expression
              symbol lexer "}"
              return $ Some s)
  <|> try (do symbol lexer "<"
              s1 <- expression
              symbol lexer "&"
              s2 <- expression
              symbol lexer ">"
              return $ s1 :& s2)
  <|> try (do symbol lexer "<"
              s1 <- expression
              symbol lexer "!"
              s2 <- expression
              symbol lexer ">"
              return $ s1 :! s2)

terminal :: Parser Expression
terminal = do
  s <- stringLiteral lexer
  return $ Terminal $ replace "<" "&#60;"
                    $ replace ">" "&#62;"
                    $ replace "&" "&amp;"
                    s

nonterminal :: Parser Expression
nonterminal = do
  s <- identifier lexer
  return $ Nonterminal s
```

One thing worthy of note is that a very basic escaping mechanism has been coded in place. While Text.Parsec handles stringLiterals quite well, outside of its regions (anything not in between double quotes already) it does not care. Here we have taken care of double black slashes, as well as HTML character escaping, allowing for proper viewing in the browser.

6 Main module

At last, the Main module. This is a very simple module that opens a file, and generates a diagram based on the EBNF syntax it parses.

```
main :: IO ()
main = do
  args <- getArgs
  mapM_ parseFile args
```

Currently, it does not support any command line arguments. It requires one compulsory argument, pointing to the location of the EBNF file to be parsed.

```
parseFile :: FilePath -> IO ()
parseFile path = do
  input <- readFile path
  result <- return (runParser grammar () path input)
  case result of
    Left err -> print err
    Right (Grammar ps) -> mapM_ outputProduction ps
```

It runs the parser on the opened file's content, and prints any parsing errors. Otherwise, it extracts each Production in the Grammar, and generates the diagrams accordingly.

```
outputProduction :: Production -> IO ()
outputProduction p@(Production name expr metas) = do
  print p
  createDirectoryIfMissing True filepath
  writeSVG (filepath ++ name ++ ".svg") pic
  writeEPS (filepath ++ name ++ ".eps") pic
  where
    pic = drawProduction p
    filepath = "./output/"
```

Each Production is saved with its title inside the output folder. If the folder doesn't exist already, it will be created upon saving. Both EPS and SVG file formats are generated at once, thanks to Wumpus.

```
drawProduction :: Production -> DPicture
drawProduction (Production name expr metas) =
    transform globalScale
  $ transform globalRotate
  $ transform globalTranslate
  $ fst
  $ drawDiagram name
  $ drawExpression expr
  where
    tau = 2 * pi
    globalTranslate = translationMatrix 0 0
    globalRotate = rotationMatrix 0
    globalScale = scalingMatrix 1 1
```

drawProduction grabs a Production and transforms it to the appropriate space. At the current stage, it is very limited; however all affine transformations are possible on the 2D plane. Scaling via the scalingMatrix is recommended, as it preserves font metrics.

```
drawExpression :: Expression -> [Component]
drawExpression e =
  case e of
    Terminal t -> [drawTerminal t]
    Nonterminal t -> [drawNonterminal t]
    Special t -> [drawSpecial t]
    OR es -> drawAlternative $ concat $ map drawExpression es
    Many e -> drawOneOrMany $ drawExpression e
    Some e -> drawZeroOrMany $ drawExpression e
    Optional e -> drawOptional $ drawExpression e
    Seq es -> drawTerminals $ concat $ map drawExpression $ concat es
    a :& b -> [drawExcept (drawExpression a) "also" (drawExpression b)]
    a :! b -> [drawExcept (drawExpression a) "not" (drawExpression b)]
```

For each Expression it's found, its type is identified, and drawn accordingly. Because this drawing is recursive in nature, it allows us to use the draw and drawBranch functions to its fullness, creating seamless diagrams.

Note: Currently, Seq is not fully supported with the line breaks. Instead, a short, horizontal rail is added, and currently wrapping around the diagram is not yet implemented. However, given the current implementation, this is not a difficult task. I simply ran out of time coding version 0.1.

7 Future works and limitations

First, the source code of EBNFRepr is available on Github[4]. Anyone with a Github account can fork the library and make modifications. A few possible improvements are noted in the README file, distributed along with the source.

A major limitation at version 0.1 is parsing. Out of all of the testing files taken from Syntrax, 2 of them failed to parse. I have no yet figured out why the parser failed, leaving this to a future maintainence issue. The other ones all draw nicely.

Another limitation at the current stage is the lack of line break and wrap around for diagrams. This feature has not been implemented due to lack of time. Also, support for command line arguments would be a bonus. Then, a standard makefile.

I noticed EBNFRepr takes more than a fraction of a second to parse a file that is only a few lines long. From speculation, a possible major limitation is that the parser can be very slow. However, since it has not yet been stress-tested in large workload, this stays as a speculation.

[4]https://github.com/SpaXe/Haector

Lastly, support for coloured railroads only has partial support. Currently this is only possible via low-level functions in Geo. Making wrappers for these functions and "upgrade" them into EBNFRepr.hs would be a great addition to the program.

8 Contact and credits

This program borrows heavily from Andrew Rock's Syntrax parser library. Most of the data structure and EBNF syntax credits go there.

The author of EBNFRepr is Xavier Ho. I can be reached at contact@xavierho.com. You're also welcome to follow me on Twitter @Xavier_Ho, and/or on Github/SpaXe. Any feedback is appreciated.