

Syntrax

Andrew Rock and David Tannock
School of Information and Communication Technology
Griffith University
a.rock@griffith.edu.au
David.Tannock@student.gu.edu.au

April 1, 2011

Abstract

Syntrax program parses Extended Backus-Naur Form (EBNF) productions and generates the corresponding syntax diagrams in Encapsulated PostScript format (EPS).

1 Using Syntrax

Syntrax¹ reads grammars in EBNF from files named as command line arguments and writes an EPS file for each production.

1.1 Building Syntrax from source code

Syntrax is written entirely in Haskell. To compile it you must have the Glasgow Haskell Compiler (GHC) installed on your system.

1.1.1 Using make

If your system is unix-like, with a `make` command:

1. Unpack the distribution file.
2. Change directory to `.../Syntrax/src`.
3. Type the command

```
make bin
```
4. Move the binary from `.../Syntrax/bin` to where you want to install it.

1.1.2 Without make

If your system does not have a `make` command:

1. Unpack the distribution file.
2. Change directory to `.../Syntrax/src`.
3. Type the command

```
ghc --make syntrax.lhs
```
4. Move the binary from `.../Syntrax/src` to where you want to install it.

¹Syntrax is based on ACESD, by David Tannock, for his Advanced Studies project in 2134CIT Programming Paradigms and Languages, 2004.

1.2 Command syntax

Use the command

```
syntrax options files
```

to process the named *files* using the specified options. Currently the options are:

- **+draw** – draw the syntax diagrams (the default);
- **-draw** – don't draw the syntax diagrams (you probably just want to check the syntax or export as XML);
- **+blue** – draw tracks, boxes and text in blue (same effect as **-rgb "0 0 1"**);
- **-blue** – draw the tracks, boxes and text in black (the default);
- **-rgb "r g b"** – draw tracks, boxes and text in the colour specified by "r g b", where $0 < r < 1$, $0 < g < 1$, $0 < b < 1$, and they are space separated and quoted to appear as one token to the shell;
- **+level** – use metadata `level="lexical"` as a cue to draw the tracks in red (the default);
- **-level** – don't use metadata `level="lexical"` as a cue to draw the tracks in red;
- **+red** – draw the tracks in red (for example to indicate no spaces allowed; same effect as **-tracksr gb "1 0 0"**);
- **-red** – draw the tracks in the same colour as the boxes and text (the default);
- **-tracksr gb "r g b"** – draw the tracks in the colour specified by "r g b");
- **+fill** – fill in the boxes with colours that distinguish terminals and nonterminals (same effect as **-termsr gb "0.8 1 0.8" -nontermsr gb "1 0.9 0.8" -specialsr gb "1 1 0.8"**);
- **-fill** – don't fill in the boxes with colours (the default);
- **-termsr gb "r g b"** – fills terminals with the colour specified by "r g b"
- **-nontermsr gb "r g b"** – fills nonterminals with the colour specified by "r g b"
- **-specialsr gb "r g b"** – fills specials with the colour specified by "r g b"
- **+some** – draw { ... }+ an alternate less compact way;
- **-some** – draw { ... }+ the compact way (the default);
- **+break** – draw long sequences as multiple rows using `\` to indicate row breaks (the default);
- **-break** – don't draw sequences as multiple rows, ignoring `\`.
- **+check** – report multiply- or un-defined nonterminal symbols (the default);
- **-check** – don't report multiply- or un-defined nonterminal symbols;
- **-xml** *outputFileName* – export all the productions as XML to file *outputFileName*.
- **-meta** *tag*[.*outputFileExtension*] – export all the metadata with the *tag* as files named for the non-terminal each production defines, with extension *outputFileExtension* or *tag* if no *outputFileExtension* is specified.
- **+dump** – print grammar parse trees (for debugging);
- **-dump** – don't print grammar parse trees (the default).

When checking or XML export are performed, all of the input files are considered as one grammar.

1.3 Input EBNF Grammar Syntax

The following EBNF sources and diagrams specify the format of the input files for `syntrax` and act as example inputs and outputs. EBNF specifications are enclosed in blue boxes. In coloured syntax diagrams:

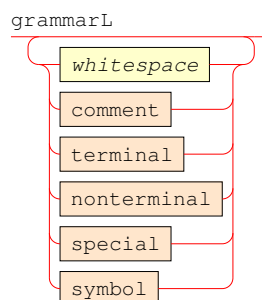
- red lines indicate that no whitespace is permitted (for lexical syntax productions);
- terminal symbols are green-filled, round-cornered rectangles;
- non-terminals are orange-filled rectangles; and
- “special” non-terminals (those specified informally) are yellow-filled rectangles.

1.3.1 Lexical syntax

```
grammarL ::= { $whitespace$ | comment | terminal
              | nonterminal | special | symbol};
```

```
purpose="With respect to its lexical syntax, a grammar is a
sequence of whitespace, comments, terminals, nonterminals,
specials and special symbols.";
level="lexical"; rank="root".
```

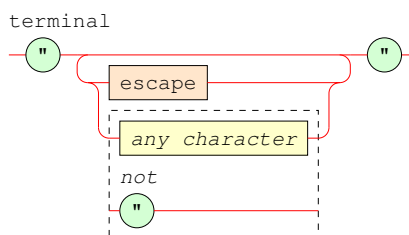
With respect to its lexical syntax, a grammar is a sequence of whitespace, comments, terminals, nonterminals, specials and special symbols.



```
terminal ::= "\"" {escape | <$any character$ ! "\""} "\"";
```

```
purpose="A {\tt terminal} symbol is a literal delimited by
double quotes. They are rendered in syntax diagrams as round
cornered boxes.";
example="\verb'\"while\"'"; example="\verb'\"\\\\\"'";
level="lexical"; rank="child".
```

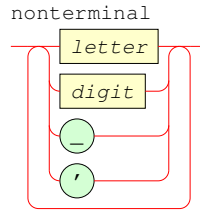
A terminal symbol is a literal delimited by double quotes. They are rendered in syntax diagrams as round cornered boxes.



```
nonterminal ::= {$letter$ | $digit$ | "_" | "'"}+;
```

```
purpose="A {\tt nonterminal} is a name defined elsewhere by a
production. They are rendered in syntax diagrams as square
cornered boxes.";
example="\verb'nonterminal'";
level="lexical"; rank="child".
```

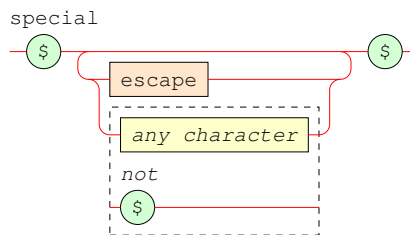
A nonterminal is a name defined elsewhere by a production. They are rendered in syntax diagrams as square cornered boxes.



```
special ::= "$" {escape | <$any character$ ! "$">} "$";
```

purpose="A {\tt special} symbol is an informally defined nonterminal delimited by dollar signs. Normally a special is rendered like a nonterminal in a square cornered box, but with italics. There are some special specials that are drawn differently: `\verb'ϵ'`; `\verb'$...$'`."; example="`\verb'$any character you want$'`"; level="lexical"; rank="child".

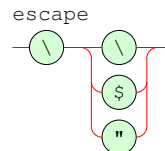
A special symbol is an informally defined nonterminal delimited by dollar signs. Normally a special is rendered like a nonterminal in a square cornered box, but with italics. There are some special specials that are drawn differently: `$epsilon$`; `$...$`.



```
escape ::= "\\\" ("\\\" | "\\$\" | "\\\"");
```

purpose="An {\tt escape} sequence permits the embedding of the special characters `\verb'\\'`, `\verb'$'` and `\verb'\"'` in terminal and special symbols."; level="lexical"; rank="child".

An `escape` sequence permits the embedding of the special characters `\`, `$` and `"` in terminal and special symbols.

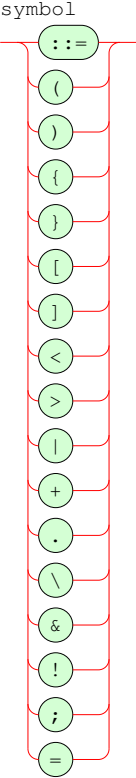


```

symbol ::=
    "::-=" | "(" | ")" | "{" | "}" | "[" | "]" |
    "<" | ">" | "|" | "+" | "." | "\" |
    "&" | "!" | ";" | "=";

purpose="The special symbols are: \verb'|',
\verb'(', \verb')', \verb'{', \verb'}',
\verb'[', \verb']', \verb'+', \verb'.',
\verb':=', \verb'\', \verb'<', \verb'>',
\verb'&', \verb'!', \verb';', and \verb'='.";
level="lexical"; rank="child".

```



The special symbols are: |, (,), {, }, [,], +, ., ::=:, \, <, >, &, !, ;, and =.

```

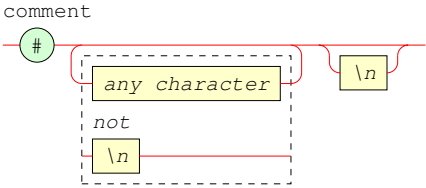
comment ::= "#" {<$any character$ ! $\n$>} [$\n$];

purpose="A {\tt comment} begins with a \verb'#' and extends to
the end of the current line. Comments are permitted anywhere
where whitespace is insignificant and is ignored like
whitespace.

This production for a comment uses a novel extension to EBNF and
to syntax diagrams. \verb'<'~$e_1$~\verb'!'~$e_2$~\verb'>'
means match $e_1$ unless it would also match $e_2$.
example="\verb'# a comment'";
level="lexical"; rank="child".

```

A comment begins with a # and extends to the end of the current line. Comments are permitted anywhere where whitespace is insignificant and is ignored like whitespace. This production for a comment uses a novel extension to EBNF and to syntax diagrams. $\langle e_1 ! e_2 \rangle$ means match e_1 unless it would also match e_2 .



1.3.2 Context-free grammar

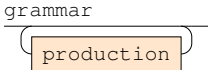
```

grammar ::= {production};

purpose="A {\tt grammar} is a sequence of productions.";
level="grammar"; rank="root".

```

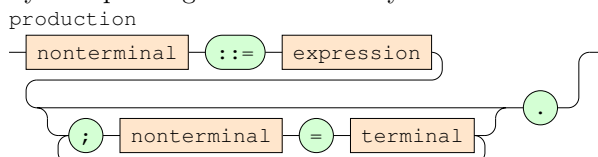
A grammar is a sequence of productions.



```
production ::= nonterminal "==" expression \
{";" nonterminal "=" terminal} ".";
```

purpose="A {\tt production} formally defines a nonterminal symbol. Normally a production consists of the name of the {\tt nonterminal} being defined, \verb'::=', and the {\tt expression} that defines that nonterminal. Optionally, meta-data may be attached. Mostly, it is ignored for the purpose of drawing syntax diagrams, but will be exported as XML. Each item of meta-data is preceded by a semicolon, named by a nonterminal and bound to a terminal value by an equals sign. The value may contain limited LaTeX markup."
example="\verb'name ::= firstName lastName.'";
level="grammar"; rank="child".

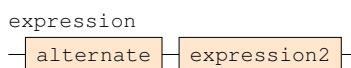
A production formally defines a nonterminal symbol. Normally a production consists of the name of the nonterminal being defined, ::=, and the expression that defines that nonterminal. Optionally, meta-data may be attached. Mostly, it is ignored for the purpose of drawing syntax diagrams, but will be exported as XML. Each item of meta-data is preceded by a semicolon, named by a nonterminal and bound to a terminal value by an equals sign. The value may contain limited LaTeX markup.



```
expression ::= alternate expression2;
```

purpose="An {\tt expression} forms the right-hand side of a production. It is specified with the auxiliary element {\tt expression2} to resolve left recursion."
level="grammar"; rank="child".

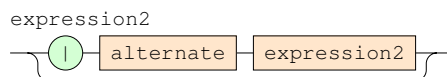
An expression forms the right-hand side of a production. It is specified with the auxiliary element expression2 to resolve left recursion.



```
expression2 ::= "|" alternate expression2 | $epsilon$;
```

purpose="How an {\tt expression} ends."
level="grammar"; rank="child".

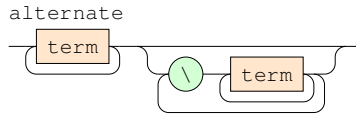
How an expression ends.



```
alternate ::= {term}+ {"\\" {term}+};
```

purpose="An {\tt alternate} is a syntactic element that may be selected from among alternatives. This production shows how terms are sequenced: normally with just whitespace between them; however a \verb'\\" placed between them is a request to break the sequence into a stack, which might help avoid a diagram that is too wide."
level="grammar"; rank="child".

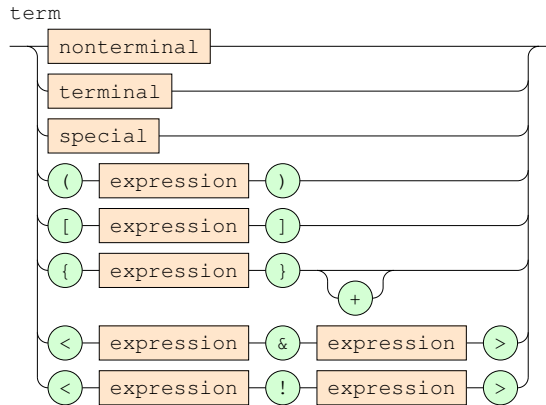
An alternate is a syntactic element that may be selected from among alternatives. This production shows how terms are sequenced: normally with just whitespace between them; however a \ placed between them is a request to break the sequence into a stack, which might help avoid a diagram that is too wide.



```
term ::=  nonterminal
        | terminal
        | special
        | "(" expression ")"
        | "[" expression "]"
        | "{" expression "} ["+" ]
        | "<" expression "&" expression ">"
        | "<" expression "!" expression ">";
```

purpose="A {\tt term} is a syntactic element that may be placed in sequence with others.";
level="grammar"; rank="child".

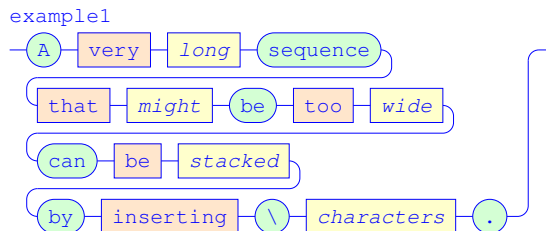
A term is a syntactic element that may be placed in sequence with others.



1.4 Special cases

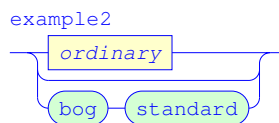
More examples highlighting special cases.

```
# stacking
example1 ::= "A" very $long$ "sequence" \ that $might$ "be"
            too $wide$ \ "can" be $stacked$ \ "by"
            inserting "\\\" $characters$ ".".
```



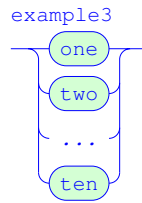
\$epsilon\$ is a special special

```
example2 ::=
    $ordinary$
    | $epsilon$
    | "bog" "standard".
```



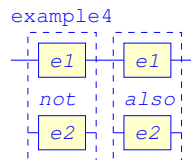
```
# $...$ is a special special
```

```
example3 ::=  
  "one"  
  | "two"  
  | $...$  
  | "ten".
```



```
# <e1 ! e2> -- match e1 but not  
#                if it also matches e2  
# <e1 & e2> -- match e1 but only  
#                if it also matches e2
```

```
example4 ::=  
  < $e1$ ! $e2$>  
  < $e1$ & $e2$>.
```



2 Implementation

2.1 Grammar data type

This module defines a data type that represents a grammar and some simple transformations.

```
module Syntrex.Grammar where
```

A **Nonterminal** is a name for a syntactic element that is defined by a production. A **Terminal** is a syntactic element that appears exactly as shown. A **Special** is an informally defined nonterminal that is not defined by a formal production.

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

An EBNF Expression is either a: **Terminal**; **Nonterminal**; **Special**; a sequence of alternatives (**OR**); a sequence of zero-or-more expressions of the same kind (**Many**); a sequence of one-or-more expressions of the same kind (**Some**); zero-or-one expressions of some kind (**Optional**); a stack of sequences of expressions (**Seq**); an expression that must also parse as some other expression (**&**); or an expression that must not also parse as some other expression (**!**).

```
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 **Production** binds a nonterminal to the EBNF expression that defines it and any metadata strings.

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

A **Grammar** is a sequence of productions.

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

getUsedNTs *p* returns the non-terminals that appear on the RHS of production *p*.

```
getUsedNTs :: Production -> [Nonterminal]
getUsedNTs (Production _ e _) = get e
  where
    get :: Expression -> [Nonterminal]
    get e = case e of
      Terminal _      -> []
      Nonterminal nt  -> [nt]
      Special _       -> []
      OR es           -> concatMap get es
      Many e         -> get e
      Some e          -> get e
      Optional e      -> get e
      Seq ess         -> concatMap (concatMap get) ess
      e :& e'         -> get e ++ get e'
      e :! e'        -> get e ++ get e'
```

2.2 Grammar lexer

This module performs the lexical analysis of an EBNF grammar.

```
module Syntrax.Lexer where

import Char

import ABR.Parser; import ABR.Parser.Lexers; import ABR.Parser.Checks

import Syntrax.Grammar

commentL :: Lexer
commentL =
  literalL '#'
  <*> (many (satisfyL (/= '\n') "")) *%> ""
  <*> (optional (literalL '\n') *%> "")
  %> " "

nonterminalL :: Lexer
nonterminalL = some (satisfyL p "letter") *%> "nonterminal"
  where
    p c = isAlpha c || isDigit c || c `elem` "_'"

escapeL :: Lexer
escapeL = literalL '\\\' *%> (literalL '\\\' <|> literalL '$' <|> literalL ''')

terminalL :: Lexer
terminalL =
  literalL '''
  *%> ((many (escapeL <|> satisfyL (/= ''') "")) *%> "terminal")
  <*> literalL '''

specialL :: Lexer
specialL =
  literalL '$'
  *%> ((many (escapeL <|> satisfyL (/= '$') "")) *%> "special")
  <*> literalL '$'

grammarL :: Lexer
grammarL = dropWhite $ nofail $ listL [
  whitespaceL, commentL, terminalL, nonterminalL, specialL,
  literalL '|' %> "symbol", literalL '(' %> "symbol",
  literalL ')' %> "symbol", literalL '{' %> "symbol",
```

```

    literalL '}' %> "symbol", literalL '[' %> "symbol",
    literalL ']' %> "symbol", literalL '+' %> "symbol",
    literalL '.' %> "symbol", tokenL "::-=" %> "symbol",
    literalL '\\\ ' %> "symbol", literalL '<' %> "symbol",
    literalL '>' %> "symbol", literalL '&' %> "symbol",
    literalL '!' %> "symbol", literalL ';' %> "symbol",
    literalL '=' %> "symbol"
]

```

2.3 Grammar parser

This module parses an EBNF grammar.

```

module Syntrax.Parse where

import ABR.Parser

import Syntrax.Grammar

termP :: Parser Expression
termP =
    tagP "nonterminal" @> (\(_,cs,_) -> Nonterminal cs)
  <|> tagP "terminal"    @> (\(_,cs,_) -> Terminal cs)
  <|> tagP "special"     @> (\(_,cs,_) -> Special cs)
  <|> literalP "symbol" "(" *> expressionP <* literalP "symbol" ")"
  <|> literalP "symbol" "[" *> expressionP <* literalP "symbol" "]"
    @> Optional
  <|> (literalP "symbol" "{" *> expressionP <* literalP "symbol" "}")
    <*> optional (literalP "symbol" "+")
    @> (\(e,ps) -> if null ps then Many e else Some e)
  <|> literalP "symbol" "<" *> expressionP <*> literalP "symbol" "&"
    *> expressionP <* literalP "symbol" ">"
    @> uncurry (:&)
  <|> literalP "symbol" "<" *> expressionP <*> literalP "symbol" "!"
    *> expressionP <* literalP "symbol" ">"
    @> uncurry (:!)

alternateP :: Parser Expression
alternateP =
    some termP
  <*> many (
    literalP "symbol" "\\\"
    *> some termP
  )
  @> (\(ts,tss) -> case filter (not . null) (ts : tss) of
    [[t]] -> t
    tss' -> Seq tss')

expressionP :: Parser Expression
expressionP = alternateP <*> expression2P @> (\(a,e') -> e' a)
  where
    expression2P :: Parser (Expression -> Expression)
    expression2P =
      literalP "symbol" "|"
      *> nofail' "alternate expected" (alternateP <*> expression2P)
      @> (\(a,e') -> e'.('join' a))
    <|> succeedA id
  where
    join :: Expression -> Expression -> Expression
    join (OR as) (OR as') = OR (as ++ as')
    join (OR as) e = OR (as ++ [e])
    join e (OR as') = OR (e : as')
    join e e' = OR [e,e']

```

```

productionP :: Parser Production
productionP =
    tagP "nonterminal"
  <*> nofail' "\"::=\" expected" (literalP "symbol" " ::=")
    *> nofail' "expression expected" expressionP
  <*> many (
    literalP "symbol" ";"
    *> (tagP "nonterminal" @> (\(_,cs,_) -> cs))
    <*> literalP "symbol" "="
    *> (tagP "terminal" @> (\(_,cs,_) -> cs))
  )
  <*> nofail' "'. ' expected" (literalP "symbol" ".")
  @> (\((_,nt,_),(e, ntts)) -> Production nt e ntts)

```

```

grammarP :: Parser Grammar
grammarP = many productionP @> Grammar

```

2.4 Drawing in EPS

This module provides support for the composition of encapsulated PostScript (EPS).

```
module Syntrax.EPS where
```

```
import ABR.Args; import ABR.Text.String
```

EPS is plain text, consisting of some header comments followed by drawing commands in PostScript. The header comments are very important as they identify the text as EPS and specify a bounding box in which the figure appears. Any drawing outside of the bounding box is clipped.

```
type EPS = String
```

A bounding Box is a tuple (*left, bottom, right, top*).

```
type Box = (Int, Int, Int, Int)
```

A PS is a sequence lines of PostScript code *in reverse order*. We build up a figure in reverse order initially to avoid a lot of use of ++.

```
type PS = [String]
```

A BPS is a figure in construction with its PS code and its bounding box.

```
type BPS = (Box, PS)
```

`epsDraw options x` renders `x` as a BPS, where `x` has a data type which is an instance of `EPSTDrawable` and `options` contains settings that might affect the rendering.

```
class EPSTDrawable a where
```

```
    epsDraw :: Options -> a -> BPS
```

`bpsToEps b` finalizes a BPS figure by reversing it and constructing the EPS header comment including the bounding box.

```
bpsToEps :: BPS -> EPS
```

```
bpsToEps ((l,b,r,t), css) = unlines $
  ["!PS-Adobe-2.0 EPSF-1.2",
   "%BoundingBox: " ++ unwords (map show [l,b,r,t]),
   "%EndComments"
  ] ++ reverse css ++ ["showpage"]
```

`psStr cs` encodes a string for inclusion in PostScript as a literal.

```

psStr :: String -> String
psStr cs = '(' : concatMap f cs ++ ")"
  where
    f c = case c of
      '\n' -> "\\n"
      '\r' -> "\\r"
      '\t' -> "\\t"
      '\b' -> "\\b"
      '\f' -> "\\f"
      '\\ ' -> "\\\\"
      '(' -> "\\("
      ')' -> "\\)"
      _ -> [c]

```

Some PostScript operators.

```

newpath = "newpath"
moveto  = "moveto"
lineto  = "lineto"
closepath = "closepath"
stroke  = "stroke"
show_   = "show"
arc     = "arc"
gsave   = "gsave"
grestore = "grestore"
translate = "translate"
setdash = "setdash"
fill    = "fill"

```

test an RGB string

```

validRGB :: String -> Bool
validRGB cs =
  let ws = words cs
      in case map isFloat ws of
    [True, True, True] -> all ((<= 1.0) . read) ws
    _                   -> False

```

2.5 Drawing syntax diagrams

This module constructs a syntax/railroad diagram from a production.

```

module Syntrax.Draw where

import ABR.Args; import ABR.Data.BSTree

import Syntrax.EPS; import Syntrax.Grammar

```

These constants define the lengths used to construct a diagram. $x_$ is an integral length. x_s is $x_$ as a string. $x_s_$ is negated $x_$ as a string. The usage of these constants is documented by the construction figures preceding drawing code that uses them.

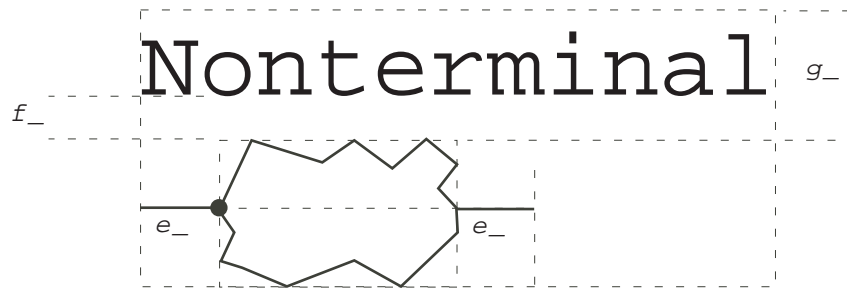
```

a_ = 8; a_s = show a_; a_s_ = show (-a_)
b_ = 5; b_s = show b_; b_s_ = show (-b_)
c_ = 6; c_s = show c_; c_s_ = show (-c_)
d_ = 3; d_s = show d_; d_s_ = show (-d_)
e_ = 8; e_s = show e_; e_s_ = show (-e_)
f_ = b_; f_s = show f_; f_s_ = show (-f_)
g_ = 13; g_s = show g_; g_s_ = show (-g_)
h_ = b_; h_s = show h_; h_s_ = show (-h_)
i_ = b_; i_s = show i_; i_s_ = show (-i_)

```

Make Production an instance of EPSTDrawable by implementing epsDraw. The option blue, if specified any way, changes the colour of the production to blue.

instance EPSTDrawable Production where



```

epsDraw options (Production nt e ms) =
  let blue = case lookupBST "blue" options of
    Just FlagPlus -> ["0 0 1 setrgbcolor"]
    -               -> case lookupBST "rgb" options of
      Just ParamMissingValue -> error "no r g b values"
      Just (ParamValue rgb) -> if validRGB rgb
        then [rgb ++ " setrgbcolor"]
        else error "bad r g b values"
    _ -> []
  options' = case lookup "level" ms of
    Just "lexical" -> case lookupBST "level" options of
      Just FlagMinus -> options
      -               -> updateBST (\x _ -> x) "red" FlagPlus options
    _ -> options
  (red, unred) = getRed options'
  ((l,b,r,t),ps) = epsDraw options' e
  l' = l - e_; b' = b; r' = max (r + e_) (l' + c_ * length nt)
  t' = t + g_
  nt' = psStr nt
  text = unwords [show l', show (t + f_), moveto, nt', show_]
  seg1 = unwords [newpath, show l', "0", moveto,
    show l, "0", lineto, stroke]
  seg2 = unwords [newpath, show r, "0", moveto,
    show (r + e_), "0", lineto, stroke]
  font = "/Courier findfont 10 scalefont setfont"
  weight = "0.4 setlinewidth"
  in ((l'-1,b'-1,r'+1,t'+1),
    ps ++ [unred, seg1, seg2, red, text, font, weight] ++ blue)

```

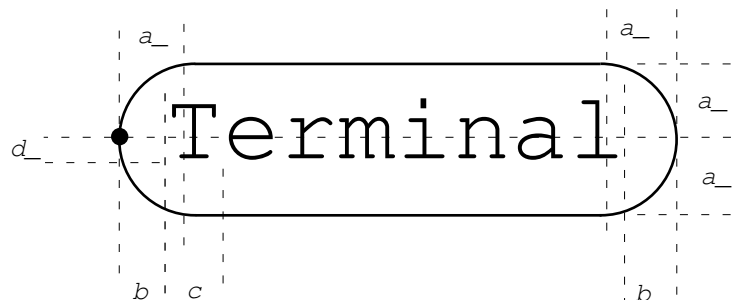
Make Expression an instance of EPSTDrawable by implementing epsDraw.

instance EPSTDrawable Expression where

```

epsDraw options e = case e of

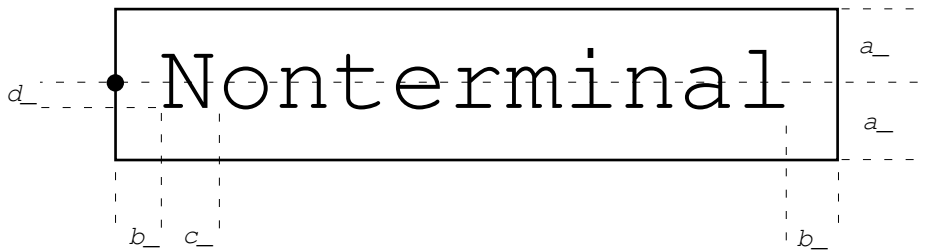
```



```

Terminal cs      ->
let width = c_ * length cs
  l = 0; b = -a_; r = width + 2 * b_; t = a_; x = r - a_;
  [ls, bs, rs, xs] = map show [l, b, r, t, x]
  fill = case lookupBST "termsrgb" options of
    Just ParamMissingValue -> error "no r g b values"
    Just (ParamValue rgb) -> if validRGB rgb
      then "gsave " ++ rgb ++ " setrgbcolor fill grestore"
      else error "bad r g b values"
  _ -> case lookupBST "fill" options of
    Just FlagPlus -> "gsave 0.83 1 0.83 setrgbcolor fill grestore"
    _ -> ""
  rect = unwords [newpath, ts, a_s, moveto,
    a_s, ls, a_s, "90", "270", arc,
    xs, bs, lineto,
    xs, ls, a_s, "270", "90", arc,
    closepath, fill, stroke]
  cs' = psStr cs
  text = unwords [b_s, d_s_, moveto, cs', show_]
in ((l,b,r,t), [text, rect])

```



```

Nonterminal cs ->
let width = c_ * length cs
  l = 0; b = -a_; r = width + 2 * b_; t = a_
  [ls, bs, rs, ts] = map show [l, b, r, t]
  fill = case lookupBST "nontermsrgb" options of
    Just ParamMissingValue -> error "no r g b values"
    Just (ParamValue rgb) -> if validRGB rgb
      then "gsave " ++ rgb ++ " setrgbcolor fill grestore"
      else error "bad r g b values"
  _ -> case lookupBST "fill" options of
    Just FlagPlus -> "gsave 1 0.9 0.8 setrgbcolor fill grestore"
    _ -> ""
  rect = unwords [newpath, ls, bs, moveto, ls, ts, lineto, rs, ts,
    lineto, rs, bs, lineto, closepath, fill, stroke]
  cs' = psStr cs
  text = unwords [b_s, d_s_, moveto, cs', show_]
in ((l,b,r,t), [text, rect])

```

A special is rendered like a nonterminal, but in italics.



There are some special specials:

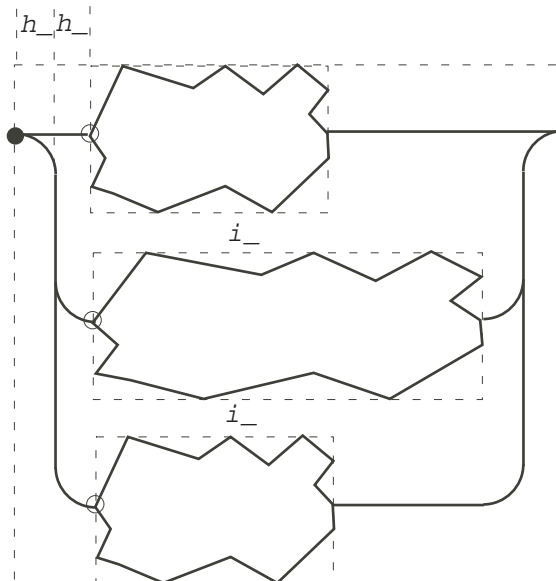
\$epsilon\$ is ϵ (the empty string) and is rendered by a line only.

\$...\$ is rendered without the box.

```

Special "epsilon" -> ((0,0,0,0),[])
Special "... " ->
  let width = c_ * length "... "
      l = 0; b = -a_; r = width + 2 * b_; t = a_
      [ls, bs, rs, ts] = map show [l, b, r, t]
      cs' = psStr "... "
      text = unwords [gsave, "/Courier-Oblique findfont 10 scalefont",
                      "setfont", b_s, d_s_, moveto, cs', show_, grestore]
  in ((l,b,r,t), [text])
Special cs ->
  let width = c_ * length cs
      l = 0; b = -a_; r = width + 2 * b_; t = a_
      [ls, bs, rs, ts] = map show [l, b, r, t]
      fill = case lookupBST "specialsr gb" options of
        Just ParamMissingValue -> error "no r g b values"
        Just (ParamValue rgb) -> if validRGB rgb
          then "gsave " ++ rgb ++ " setrgbcolor fill grestore"
          else error "bad r g b values"
      _ -> case lookupBST "fill" options of
        Just FlagPlus -> "gsave 1 1 0.8 setrgbcolor fill grestore"
        _ -> ""
      rect = unwords [newpath, ls, bs, moveto, ls, ts, lineto, rs, ts,
                      lineto, rs, bs, lineto, closepath, fill, stroke]
      cs' = psStr cs
      text = unwords [gsave, "/Courier-Oblique findfont 10 scalefont",
                      "setfont", b_s, d_s_, moveto, cs', show_, grestore]
  in ((l,b,r,t), [text, rect])

```



```

OR es ->
  let bpss = map (epsDraw options) es
      ((_,b',r',t),alt0) = head bpss
      l = 0;
      r = maximum (map (\((_,_,r,_),_) -> r) bpss) + 4 * h_
      b = sum (map (\((_,b,_),t) -> b - t) bpss)
          - i_ * (length bpss - 1) + t
      drop = b - (\((_,b,_),_) -> b) (last bpss) + h_
      [ls,bs,rs,ts] = map show [l,b,r,t]
      trans0 = unwords [gsave, show (2 * h_), ls, translate]
      (red, unred) = getRed options
      segL = unwords [red, newpath,
                      h_s, (show drop), moveto,

```

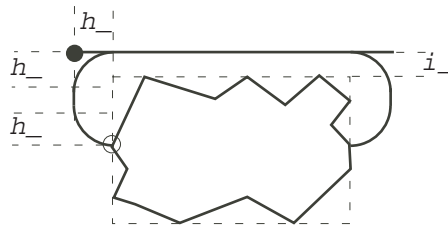
```

ls, h_s_, h_s, "0", "90", arc,
show (2 * h_), ls, lineto,
stroke, unred]
segR = unwords [red, newpath,
show (r' + 2 * h_), ls, moveto,
rs, h_s_, h_s, "90", "180", arc,
show (r - h_), (show drop), lineto,
stroke, unred]
alts [] _ = []
alts ((l',b',r',t'), ps) : rest) b = [grestore] ++
alts rest b' ++
[unwords [red, newpath,
"0", h_s, h_s, "180", "270", arc,
stroke, newpath,
show r', "0", moveto,
show (r - 4 * h_), h_s, h_s, "270", "360", arc,
stroke, unred]
] ++ ps ++
[unwords ["0", show (b - i_ - t'), translate],
gsave]
in ((l,b,r,t), [segL, segR, grestore] ++ alts (tail bpss) b' ++ alt0
++ [trans0])

```

Many can be drawn two ways:

1. The simplest has a loop of track, but note that it goes through the item backwards. For sequences this is wrong.



2. In the case of sequences many is drawn as optionally-some.

```

Many e      ->
let (red, unred) = getRed options
anySequence :: Expression -> Bool
anySequence e = case e of
  Terminal _ -> False
  Nonterminal _ -> False
  Special _ -> False
  OR es -> any anySequence es
  Many e -> anySequence e
  Some e -> anySequence e || (case lookupBST "some" options of
    Just FlagPlus -> True
    - -> False
  )
Optional e -> anySequence e
Seq _ -> True
e :& e' -> anySequence e || anySequence e'
e :! e' -> anySequence e || anySequence e'
simpleMany =
let ((l,b,r,t), ps) = epsDraw options e
    drop = max (2 * h_) (i_ + t)
    l' = 0; b' = b - drop; r' = r + 2 * h_; t' = 0
    [ls, bs, rs, ts] = map show [l', b', r', t']
    trans = unwords [gsave, h_s, show (-drop), translate]

```

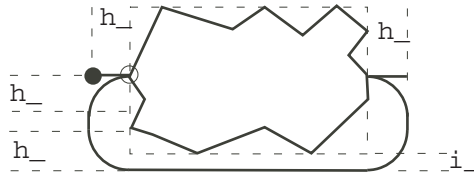
```

path1 = unwords [red, newpath,
  ls, ts, moveto,
  rs, ts, lineto,
  stroke, unred]
path2 = unwords [red, newpath,
  h_s, ts, moveto,
  h_s, h_s_, h_s, "90", "180", arc,
  h_s, show (h_ - drop), h_s, "180", "270", arc,
  stroke, unred]
path3 = unwords [red, newpath,
  show (r + h_), show (-drop), moveto,
  show (r + h_), show (h_ - drop), h_s, "270", "360", arc,
  show (r + h_), h_s_, h_s, "0", "90", arc,
  stroke, unred]
in ((l',b',r',t'), [path1, path2, path3, grestore] ++ ps ++ [trans])
in if anySequence e
then epsDraw options (Optional (Some e))
else simpleMany

```

Some can be drawn two ways:

1. The simplest has a loop of track (the default).

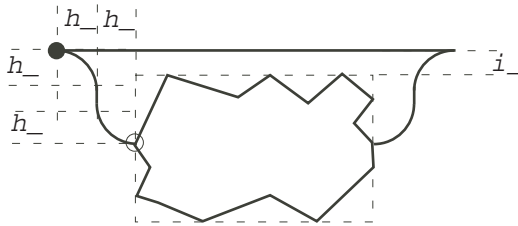


2. The second (optionally selected) is to make a sequence of one followed by many.

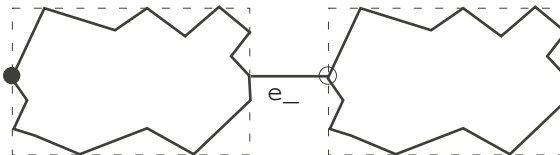
```

Some e      -> case lookupBST "some" options of
Just FlagPlus -> epsDraw options (Seq [[e, Many e]])
-           ->
let (red, unred) = getRed options
((l,b,r,t), ps) = epsDraw options e
trans = unwords [gsave, h_s, "0", translate]
l' = 0; b' = b - i_; r' = r + 2 * h_; t' = t
path1 = unwords [red, newpath,
  "0", "0", moveto,
  h_s, "0", lineto,
  stroke, unred]
path2 = unwords [red, newpath,
  show (r + h_), "0", moveto,
  show r', "0", lineto,
  stroke, unred]
path3 = unwords [red, newpath,
  h_s, "0", moveto,
  h_s, h_s_, h_s, "90", "180", arc,
  "0", show (b' + h_), lineto,
  h_s, show (b' + h_), h_s, "180", "270", arc,
  h_s, show b', show (r' - h_), show b', lineto,
  show (r' - h_), show (b' + h_), h_s, "270", "360", arc,
  show r', show (b' + h_), show r', h_s_, lineto,
  show (r + h_), h_s_, h_s, "0", "90", arc,
  stroke, unred]
in ((l',b',r',t'), [path1, path2, path3, grestore] ++ ps ++ [trans])

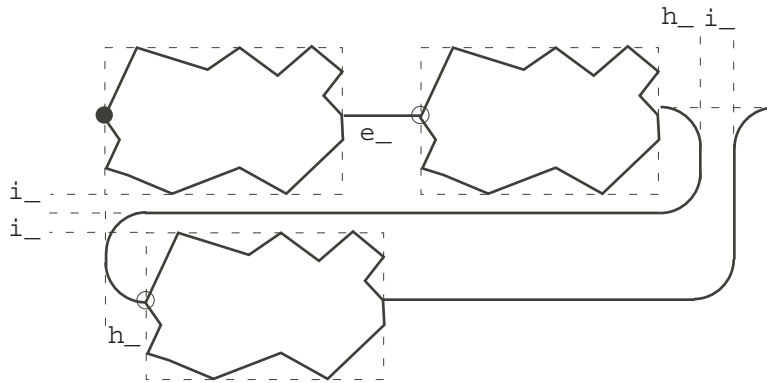
```



```
Optional e    ->
let (red, unred) = getRed options
  ((l,b,r,t), ps) = epsDraw options e
  drop = max (2 * h_) (i_ + t)
  l' = 0; b' = b - drop; r' = r + 4 * h_; t' = 0
  [ls, bs, rs, ts] = map show [l', b', r', t']
  trans = unwords [gsave, show (2 * h_), show (-drop), translate]
  path1 = unwords [red, newpath,
    h_s, h_s_, moveto,
    "0", h_s_, h_s, "0", "90", arc,
    rs, "0", lineto,
    rs, h_s_, h_s, "90", "180", arc,
    stroke, unred]
  path2 = unwords [red, newpath,
    h_s, h_s_, moveto,
    show (2 * h_), show (h_ - drop), h_s, "180", "270", arc,
    stroke, unred]
  path3 = unwords [red, newpath,
    show (r + 2 * h_), show (-drop), moveto,
    show (r + 2 * h_), show (h_ - drop), h_s, "270", "360", arc,
    show (r + 3 * h_), h_s_, lineto,
    stroke, unred]
in ((l',b',r',t'), [path1, path2, path3, grestore] ++ ps ++ [trans])
```



```
Seq [e:es]    ->
let ((l,b,r,t),ps) = epsDraw options e
in case es of
  [] -> ((l,b,r,t),ps)
  - ->
    let (red, unred) = getRed options
      ((l',b',r',t'), ps') = epsDraw options (Seq [es])
      l'' = l; b'' = min b b'; r'' = r + r' + e_; t'' = max t t'
      seg = unwords [red, newpath, show r, "0 moveto", show (r + e_),
        "0", lineto, stroke, unred]
      trans = unwords [gsave, show (r + e_), "0 translate"]
    in ((l'', b'', r'', t''),
      ps ++ [seg, grestore] ++ ps' ++ [trans])
```



```

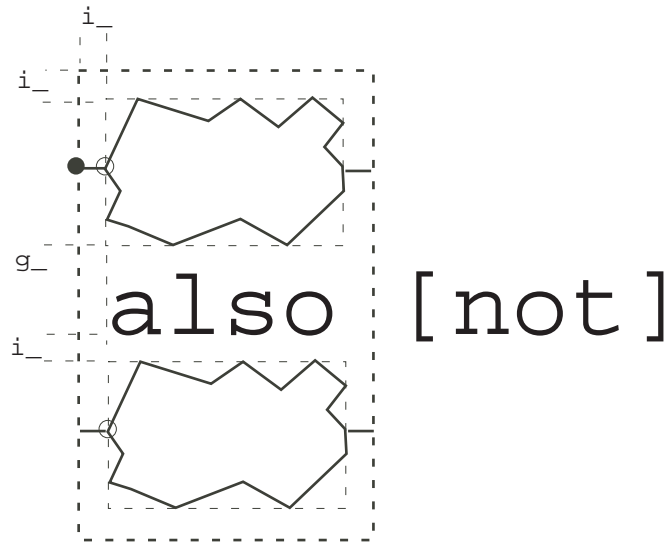
Seq ess -> if lookupFlag "break" options True
  then let (red, unred) = getRed options
        (bps : bpss) = map (epsDraw options . Seq . (: [])) ess
        ((l,b,r,t),ps) = bps
        ((l',b',r',t'),ps') = last bpss
        maxr = max r $ maximum $ map (\((_,_,r,_),_) -> r + h_) bpss
        l'' = 0
        t'' = t
        r'' = maxr + 2 * h_ + i_
        drops (((_,b,_,t),_) : ((l',b',r',t'),ps') : bpss) d =
          let bumroom = min (b - i_) (negate i_)
              headroom = max (t' + i_) (2 * h_)
              thisDrop = bumroom - headroom
          in (d + thisDrop) : drops (((l',b',r',t'),ps') : bpss) (d + thisDrop)
        drops _ _ = []
        ds = drops (bps : bpss) 0
        d' = last ds
        b'' = d' + (\((_,b,_,_),_) -> b) (last bpss)
        trans1 (((l,b,r,t),ps),d) =
          let headroom = max (t + i_) (2 * h_)
              path1 = unwords [
                red, newpath,
                "0", show headroom, moveto,
                "0", show (headroom - h_), h_s, "90 180 arc",
                h_s_, h_s, lineto,
                "0", h_s, h_s, "180 270 arc",
                stroke, unred]
          in [grestore, path1] ++ ps ++
            [unwords [gsave, h_s, show d, translate]]
        trans2 (((l,b,r,t),ps),d) =
          let path3 = unwords [
                red, newpath,
                "0", show (b - i_), moveto,
                show r, show (b - i_), lineto,
                show r, show (b + i_ - h_), h_s, "270 360 arc",
                show r, h_s_, h_s, "0 90 arc",
                stroke, unred]
          in [grestore, path3] ++
            [unwords [gsave, h_s, show d, translate]]
        path2 = unwords [red, newpath,
          show (r' + h_), show d', moveto,
          show (maxr + i_), show d', lineto,
          show (maxr + i_), show (d' + h_), h_s, "270 360 arc",
          show (maxr + i_ + h_), h_s_, lineto,
          stroke, newpath,
          show (maxr + i_ + 2 * h_), h_s_, h_s, "90 180 arc",
          stroke, unred]

```

```

    path4 = unwords [
      red, newpath,
      h_s, show (b - i_), moveto,
      show r, show (b - i_), lineto,
      show r, show (b + i_ - h_), h_s, "270 360 arc",
      show r, h_s_, h_s, "0 90 arc",
      stroke, unred]
  in ((l'', b'', r'', t''), ps
    ++ concatMap trans1 (zip bpss ds)
    ++ concatMap trans2 (init (zip bpss ds))
    ++ [path2, path4]
  )
else epsDraw options $ Seq [concat ess]

```



```

e :& e'      ->
let (red, unred) = getRed options
    ((l,b,r,t),ps) = epsDraw options e
    ((l',b',r',t'),ps') = epsDraw options e'
    drop = b - g_ - i_ - t'
    trans = unwords [gsave, i_s, "0", translate]
    trans' = unwords [gsave, i_s, show drop, translate]
    l'' = 0
    r'' = max r r' + 2 * i_
    t'' = t + i_
    b'' = drop + b' - i_
    path1 = unwords [red, newpath,
      "0", "0", moveto,
      i_s, "0", lineto,
      stroke, unred]
    path2 = unwords [red, newpath,
      show (r + i_), "0", moveto,
      show r'', "0", lineto,
      stroke, unred]
    path3 = unwords [red, newpath,
      "0", show drop, moveto,
      i_s, show drop, lineto,
      stroke, unred]
    path4 = unwords [red, newpath,
      show (r' + i_), show drop, moveto,
      show r'', show drop, lineto,
      stroke, unred]
    path5 = unwords [gsave, "[3 3] 0 setdash", newpath,

```

```

        show l'', show b'', moveto,
        show r'', show b'', lineto,
        show r'', show t'', lineto,
        show l'', show t'', lineto,
        closepath,
        stroke, grestore]
    text = unwords [gsave, "/Courier-Oblique findfont 10 scalefont",
        "setfont", i_s, show (b - g_), moveto, "(also)", show_, grestore]
in ((l'',b'',r'',t''), [path1, path2, path3, path4, path5, text, grestore]
    ++ ps' ++ [trans', grestore] ++ ps ++ [trans])
e :! e'      ->
let (red, unred) = getRed options
    ((l,b,r,t),ps) = epsDraw options e
    ((l',b',r',t'),ps') = epsDraw options e'
    drop = b - g_ - i_ - t'
    trans = unwords [gsave, i_s, "0", translate]
    trans' = unwords [gsave, i_s, show drop, translate]
    l'' = 0
    r'' = max r r' + 2 * i_
    t'' = t + i_
    b'' = drop + b' - i_
    path1 = unwords [red, newpath,
        "0", "0", moveto,
        i_s, "0", lineto,
        stroke, unred]
    path2 = unwords [red, newpath,
        show (r + i_), "0", moveto,
        show r'', "0", lineto,
        stroke, unred]
    path3 = unwords [red, newpath,
        "0", show drop, moveto,
        i_s, show drop, lineto,
        stroke, unred]
    path4 = unwords [red, newpath,
        show (r' + i_), show drop, moveto,
        show r'', show drop, lineto,
        stroke, unred]
    path5 = unwords [gsave, "[3 3] 0 setdash", newpath,
        show l'', show b'', moveto,
        show r'', show b'', lineto,
        show r'', show t'', lineto,
        show l'', show t'', lineto,
        closepath,
        stroke, grestore]
    text = unwords [gsave, "/Courier-Oblique findfont 10 scalefont",
        "setfont", i_s, show (b - g_), moveto, "(not)", show_, grestore]
in ((l'',b'',r'',t''), [path1, path2, path3, path4, path5, text, grestore]
    ++ ps' ++ [trans', grestore] ++ ps ++ [trans])

getRed :: Options -> (String, String)
getRed options = case lookupBST "red" options of
    Just (FlagPlus) -> ("gsave 1 0 0 setrgbcolor", grestore)
    _                 -> case lookupBST "tracksrGB" options of
        Just ParamMissingValue -> error "missing r g b value"
        Just (ParamValue rgb) -> if validRGB rgb
            then ("gsave " ++ rgb ++ " setrgbcolor", grestore)
            else error "bad r g b values"
    _ -> ("", "")

```

2.6 Exporting as XML

This module exports a grammar as XML.

```
{-# LANGUAGE FlexibleInstances, TypeSynonymInstances, OverlappingInstances #-}
module Syntrax.XML where

import System.IO

import ABR.Args; import ABR.Data.BSTree; import ABR.Text.Markup

import Syntrax.Grammar

exportXML options ps exports productions ps to the file specified in options.

exportXML :: Options -> [Production] -> IO ()
exportXML options ps = do
  let path = lookupParam "xml" options "NO-FILE-SPECIFIED"
      h <- openFile path WriteMode
      hPutStrLn h "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
      hPutStrLn h "<grammar>"
      xml h options 1 ps
      hPutStrLn h "</grammar>"
      hClose h
```

Class XML overloads xml. `xml h options i x` writes `x` to handle `h` with indent level `i` according to `options`.

```
class XML a where
  xml :: Handle -> Options -> Int -> a -> IO ()
  xml h options i a = do
    indent h options i
    hPutStrLn h "<undefined />"
```

`indent h options` writes enough spaces to handle `h` to indent to level `i` according to `options`.

```
indent :: Handle -> Options -> Int -> IO ()
indent h _ i = hPutStr h $ replicate (3 * i) ' '
```

Instances:

```
instance XML a => XML [a] where
  xml h options i xs = mapM_ (xml h options i) xs
```

```
instance XML Production where
  xml h options i (Production nt e nnts) = do
    indent h options i
    hPutStrLn h $ "<production name=\"\" ++ nt ++ \">"
    xml h options (i + 1) e
    xml h options (i + 1) nnts
    indent h options i
    hPutStrLn h "</production>"
```

```
instance XML Expression where
  xml h options i e = do
    indent h options i
    hPutStr h "<expression type=\"\"
  case e of
    Terminal t      -> do
      hPutStr h "terminal\>"
      xml h options (i + 1) t
    Nonterminal nt -> do
      hPutStr h $ "nonterminal\>"
      xml h options (i + 1) nt
```

```

Special s      -> do
  hPutStr h "special\>"
  xml h options (i + 1) s
OR es         -> do
  hPutStrLn h "or\>"
  xml h options (i + 1) es
  indent h options i
Many e        -> do
  hPutStrLn h "many\>"
  xml h options (i + 1) e
  indent h options i
Some e        -> do
  hPutStrLn h "some\>"
  xml h options (i + 1) e
  indent h options i
Optional e    -> do
  hPutStrLn h "optional\>"
  xml h options (i + 1) e
  indent h options i
Seq ess       -> do
  hPutStrLn h "sequence\>"
  mapM_ (\es -> do
    indent h options (i+1)
    hPutStrLn h "<row>"
    xml h options (i + 2) es
    indent h options (i+1)
    hPutStrLn h "</row>"
  ) ess
  indent h options i
e :& e'       -> do
  hPutStrLn h "also\>"
  xml h options (i + 1) e
  xml h options (i + 1) e'
  indent h options i
e :! e'       -> do
  hPutStrLn h "not\>"
  xml h options (i + 1) e
  xml h options (i + 1) e'
  indent h options i
hPutStrLn h "</expression>"

```

```

instance XML (Nonterminal, Terminal) where
  xml h options i (nt, t) = do
    indent h options i
    hPutStr h $ "<meta name=\"" ++ nt ++ "\">"
    hPutStr h $ latex2html t
    hPutStrLn h "</meta>"

```

```

instance XML String where
  xml h options i cs = case cs of
    ""      -> return ()
    c : cs' -> case c of
      ''' -> hPutStr h $ makeHTMLSafe cs
      '$' -> hPutStr h $ makeHTMLSafe cs
      _   -> hPutStr h $ makeHTMLSafe cs

```

2.7 Syntrex main module

This is the main module for the `syntrex` command-line tool.

```

module Main (main) where

```

```

import System; import IO; import List; import Monad

import ABR.Parser; import ABR.Args; import ABR.Control.Check
import ABR.Data.BSTree; import ABR.Parser.Checks

import Syntrax.Grammar; import Syntrax.Lexer; import Syntrax.Parse
import Syntrax.EPS; import Syntrax.Draw; import Syntrax.XML
import Syntrax.Meta

main :: IO ()
main = do
  args <- getArgs
  let (options,files) =
        findOpts [FlagS "draw", FlagS "blue", FlagS "check", FlagS "some",
                  FlagS "break", FlagS "red", FlagS "fill", FlagS "dump",
                  ParamS "rgb", ParamS "tracksrgb", ParamS "termsrgb",
                  ParamS "nontermsrgb", ParamS "specialsrgb",
                  ParamS "xml", QueueS "meta", FlagS "level"] args
  mgs <- mapM (load options) files
  let ps = concat [ps | Just (Grammar ps) <- mgs]
  case lookupBST "check" options of
    Just FlagMinus -> return ()
    -               -> do
      let nts = [nt | Production nt _ _ <- ps]
          dnts = nub [nt | (nt:nts') <- tails nts, nt `elem` nts']
          nts' = nub $ concatMap getUsedNTs ps
          unts = nts' \\ nts
      unless (null dnts) (do
        hPutStrLn stderr "Multiply defined non-terminals:"
        mapM_ (hPutStrLn stderr . ('\t' :)) dnts
      )
      unless (null unts) (do
        hPutStrLn stderr "Undefined non-terminals:"
        mapM_ (hPutStrLn stderr . ('\t' :)) unts
      )
  case lookupBST "draw" options of
    Just FlagMinus -> return ()
    -               -> mapM_ (drawProduction options) ps
  case lookupBST "xml" options of
    Just _ -> exportXML options ps
    -       -> return ()
  case lookupBST "meta" options of
    Just _ -> exportMeta options ps
    -       -> return ()

```

load *f* reads and attempts to parse the grammar in file named *f*.

```

load :: Options -> FilePath -> IO (Maybe Grammar)
load options f = do
  grammar <- readFile f
  case checkParse (total grammarL) (total grammarP) grammar of
    CheckPass g   -> do
      case lookupBST "dump" options of
        Just FlagPlus -> do
          putStrLn $ "### file: " ++ f ++ " parse tree:"
          print g
        _ -> return ()
      return $ Just g
    CheckFail msg -> do
      hPutStrLn stderr $ "Syntax error in file: " ++ f
      hPutStr stderr msg
      return Nothing

```

process *options p* generates an EPS file for production *p*, using the command line *options*.

```
drawProduction :: Options -> Production -> IO ()
drawProduction options p@(Production nt _) =
  writeFile (nt ++ ".eps") $ bpsToEps $ epsDraw options p
```