Note [Parsing ">"]  
  
When Java was originally designed, there were no type parameters. For example,  
there was no ArrayList<Integer>, there was just ArrayList. All containers  
stored Objects, and you were required to cast every time you retrieved an  
element from a container. Life was bad.  

However, when generics were introduced in Java 5, the new syntax, using  
< and >, conflicted with existing usages of < and > (as comparison operators).  
For example, does "foo<bar.baz" start an expression, comparing foo against  
bar.baz? Or is it the beginning of a parameterized type foo, where bar.baz  
is the parameter? It’s impossible to know until we find the >, which can  
be arbitrarily far away. Real Java parsers go through machinations to get  
around this ambiguity, but we don’t want the parser here to be quite so  
difficult. Instead, we use the following rules:  

Iff "<" is preceded by whitespace, it is a comparison operator.  
Iff "<" is preceded by non-whitespace, it begins a type argument.  

This is a bit silly to use whitespace in this way, but it’s very convenient.  
According to the documentation for Alex, each of the lexing rules tries  
to consume as much of the input as possible. So, even though the rules  
for '@WhiteSpace' and '@WhiteSpace "<"' overlap, the second one will always  
win. Along similar lines, the rule for plain ""<"' will never fire if the  
symbol is preceded by whitespace. We can thus detect these two cases easily.  

We must similarly define extra rules for <=, <<, and <<=, because otherwise  
the " <" that might precede them might get mistaken for a LessThan.  

Naturally, the two meanings of < correspond to different tokens.  

Interestingly, there is no such challenge with >, because we know whether  
or not we’ve recently passed a < that is the beginning of a type argument.  
However (and unlike real Java implementations), we fail on something like  
'ArrayList<List<Bool>>', because the '>>' at the end is lexed as a right-  
shift operator. It wouldn’t be hard to fix this, actually: we could refuse,  
for example, to lex a right-shift after the beginning of a type argument  
(until we see the end of the type argument). Alex supports stateful lexing,  
so this should be doable without terribly much fuss. One challenge is that  
it would have to track the number of type-argument beginnings so it could  
be sure when the type argument is truly over. Though Alex can be stateful,  
it doesn’t internally support an unbounded number of states. The solution  
would be to store the type-argument depth in a monad and use an Alex  
predicate, but that’s way more elaborate than I want here.

{-- Module : Language.Java.Lexer --}
import Language.Java.Panic
import Language.Java.Token
import Language.Java.Literal
import Language.Java.Monad
import Language.Java.Char

-- This imports various automatic processing utilities. See the alex manual.
%wrapper "monadUserState"

$InputCharacter = . # \r
$AnyChar = [.\n]

$JavaLetter = [A-Za-z_\$]     -- S3.8
$JavaLetterOrDigit = [0-9$JavaLetter]

@LineTerminator = [\r\n] | \r
                   -- newlines (S3.4)
@WhiteSpace     = \                               -- whitespace (S3.6)
                 | \t
                 | \f
                 | @LineTerminator

-- Comments, S3.7
@CommentElement    = ($AnyChar # \*) | \
                      ($AnyChar # /)
@CommentTerminator = \
                      
@CommentTail       = @CommentElement* @CommentTerminator

@TraditionalComment = "/*" @CommentTail

@EndOfLineComment = "//" $InputCharacter*

@Comment        =  @TraditionalComment
                 |  @EndOfLineComment

-- Integer literals, S3.10.1
$IntegerTypeSuffix = [lL]
$Digit             = 0-9
$NonZeroDigit      = 1-9
@DecimalNumeral    = 0
                    | $NonZeroDigit
                    | $NonZeroDigit ($Digit | _)* $Digit

$HexDigit   = [0-9a-fA-F]
@HexDigits  = $HexDigit
             | $HexDigit ($HexDigit | _)* $HexDigit
@HexNumeral = 0 [xX] @HexDigits

$OctalDigit   = [0-7]
@OctalNumeral = 0 ($OctalDigit | _)* $OctalDigit

$BinaryDigit   = [01]
@BinaryDigits  = $BinaryDigit
                | $BinaryDigit ($BinaryDigit | _)* $BinaryDigit
@BinaryNumeral = 0 [bB] @BinaryDigits

-- Floating point literals, S3.10.2
@Digits          = $Digit
                    | $Digit ($Digit | _)* $Digit
$FloatTypeSuffix = [fFdD]
@SignedInteger   = [\+\-]? @Digits
@ExponentPart    = [eE] @SignedInteger

@DecimalFloatingPointLiteral = @Digits \. @Digits? @ExponentPart? $FloatTypeSuffix?

Java.x

145:  @HexSignificand                  = @HexNumeral \.?
146:          | 0 [xX] @HexDigits? \. @HexDigits
147:  @BinaryExponent                  = [pP] @SignedInteger
148:  @HexadecimalFloatingPointLiteral = @HexSignificand @BinaryExponent $FloatTypeSuffix?
149:
150:  -- Escape sequences, S3.10.6
151:  @EscapeSequence = \ \ ( [bntfr"\'\"]
152:          | $OctalDigit
153:          | $OctalDigit $OctalDigit
154:          | [0-3] $OctalDigit $OctalDigit )
155:
156:  -- Character literals, S3.10.4
157:  $SingleCharacter  = $InputCharacter # ["\"
158:  @CharacterLiteral = ' $SingleCharacter '
159:          | ' @EscapeSequence '
160:
161:  -- String literals, S3.10.5
162:  @StringCharacter = $InputCharacter # ["\"
163:          | @EscapeSequence
164:
165:  -- Identifiers, S3.8
166:  $JavaLetter $JavaLetterOrDigit*     { tokenS identifier }  
167:
168:  -- Integer literals, S3.10.1
169:  -- IRREG: Despite the JLS’s exhortation, we do *not* fail on overflowing literals.
170:  @DecimalNumeral                     { tokenS (LiteralT . IntL . read . stripUnderscores) }
171:  @DecimalNumeral $IntegerTypeSuffix  { tokenS (LiteralT . LongL . read . stripUnderscores . 
172:          | init) }
173:
174:  @HexNumeral                         { tokenS (LiteralT . IntL . read . stripUnderscores) }
175:  @HexNumeral $IntegerTypeSuffix      { tokenS (LiteralT . LongL . read . stripUnderscores . 
176:          | init) }
177:
178:  @OctalNumeral                       { tokenS (LiteralT . IntL . read . insertOctalO . stri 
179:          | pUnderscores) }
180:  @OctalNumeral $IntegerTypeSuffix    { tokenS (LiteralT . LongL . read . insertOctalO . str 
181:          | ipUnderscores . init) }
182:
183:  @BinaryNumeral                      { tokenS (LiteralT . IntL . readBin . stripUnderscores 
184:          ) }
185:  @BinaryNumeral $IntegerTypeSuffix   { tokenS (LiteralT . LongL . readBin . stripUnderscore 
186:          s . init) }
187:
188:  @DecimalFloatingPointLiteral        { tokenS lexFloatLiteral }
189:  @HexadecimalFloatingPointLiteral    { tokenS \($s -> unimplemented $ "lex hexademical liter 
190:        als like " ++ s) }
191:
192:  @CharacterLiteral                   { tokenS (LiteralT . CharL . charHtoJ . read . insertO 
193:          ctalOEscapes) }
194:
195:  @StringLiteral                      { tokenS (LiteralT . StringL . stringHtoJ . read . ins 
196:          ertOctalOEscapes) }
197:
198:  -- Separators, S3.11
199:  "(" { sep LParenS }  
200:  ")"  { sep RParenS }  
201:  "["  { sep LBraceS }  
202:  "]"  { sep RBraceS }  
203:  "}"  { sep LBracketS }  
204:  "]"  { sep RBracketS }  
205:  ","  { sep SemicolonS }
Java

208: "," { sep CommaS }
209: "." { sep DotS }
210: "..." { sep EllipsisS }
211: "@" { sep AtS }
212: "::" { sep DoubleColonS }
213: "<" { sep LAngleBracketS } -- See Note [Parsing "<"]
214:
215: -- Operators, S3.12
216: "=" { op AssignO }
217: "==" { op EqualsO }
218: "+=" { op PlusEqualO }
219: "+" { op PlusO }
220: ">" { op GreaterO }
221: ">=" { op GreaterEqualO }
222: "-" { op MinusO }
223: "~-" { op MinusEqualO }
224: @WhiteSpace "<" { op LessO } -- See Note [Parsing "<"]
225: "<=" { op LessEqualO }
226: @WhiteSpace "<=" { op LessEqualO } -- See Note [Parsing "<"]
227: "+" { op PlusO }
228: "*" { op TimesO }
229: "*/" { op NotO }
230: "/=" { op NotEqualO }
231: "!/" { op DivideO }
232: "\=/" { op DivideEqualO }
233: "~" { op BinaryNotO }
234: "&&" { op AndO }
235: "&" { op BinaryAndO }
236: "&=" { op BinaryAndEqualO }
237: "|" { op QuestionO }
238: "><" { op OrO }
239: "||" { op BinaryOrO }
240: "+=" { op BinaryOrEqualO }
241: "+=" { op ColonO }
242: "++=" { op PlusPlusO }
243: "+~" { op BinaryXorO }
244: "+~=" { op BinaryXorEqualO }
245: "+>-" { op ArrowO }
246: "~-=" { op MinusMinusO }
247: "/~" { op ModulusO }
248: "/~=" { op ModulusEqualO }
249: "<<" { op ShiftLeftO }
250: @WhiteSpace "<<<" { op Shift LeftO } -- See Note [Parsing "<"]
251: "<<=" { op ShiftLeftEqualO }
252: @WhiteSpace "<<<<" { op ShiftLeftEqualO } -- See Note [Parsing "<"]
253: ">>>
254: ">>=" { op ShiftRightO }
255: ">>>=" { op ShiftRightZeroExtensionO }
256: ">>>=" { op ShiftRightZeroExtensionEqualO }
257:
258: -- This brace marks the start of Haskell code that will process the above
259: -- directives
260: {
261: 
262:  --------------------------------------------------------------------------
263:  -- This region is all internal Alex plumbing. Haskell experts only!
264:  --------------------------------------------------------------------------
265:  type AlexUserState = [Token] -- in reverse order for easy consing
266:  alexUpdUserState :: ([Token] -> [Token]) -> Alex ()
267:  alexUpdUserState upd = do
268:    st <- alexGetUserState
269:    alexSetUserState (upd st)
270:  alexGetString :: AlexInput -> Int -> String
271:  alexGetString (_,_,_,str) len = take len str
272:  tokenS :: (String -> Token) -> AlexInput -> Int -> Alex ()
273:  tokenS f input len = do
274:    let tok = f (alexGetString input len)
275:    alexUpdUserState (tok :)
276: 

alexMonadScan -- and continue

This is called when the end of file is reached
alexEOF :: Alex ()
alexEOF = do
(AlexPn _ line column, _, remaining_bytes, remaining_chars) <- alexGetInput
if null remaining_bytes && null remaining_chars
  then return () -- success
  else alexError $ "Unexpected end of file at line " ++ show line ++ ", column " ++ show column

-- the initial state
alexInitUserState :: [Token]
alexInitUserState = []

-- Utility functions used during lexing

-- Convert a String into a Token, recognizing keywords.
identifier :: String -> Token
  -- the literals are easy enough to handle by direct pattern matching
identifier "true"  = LiteralT (BooleanL True)
identifier "false" = LiteralT (BooleanL False)
identifier "null"  = LiteralT NullL
identifier str
  = case M.lookup str keywordMap of
      Nothing -> IdentifierT str -- not a keyword
      Just k  -> KeywordT k      -- a keyword

-- See S3.9
keywordMap :: M.Map String Keyword
keywordMap = M.fromList
  [ ("abstract",     AbstractK)
  , ("assert",       AssertK)
  , ("boolean",      BooleanK)
  , ("break",        BreakK)
  , ("case",         CaseK)
  , ("catch",        CatchK)
  , ("char",         CharK)
  , ("class",        ClassK)
  , ("const",        ConstK)
  , ("continue",     ContinueK)
  , ("default",      DefaultK)
  , ("do",           DoK)
  , ("double",       DoubleK)
  , ("else",         ElseK)
  , ("enum",         EnumK)
  , ("extends",      ExtendsK)
  , ("final",        FinalK)
  , ("finally",      FinallyK)
  , ("float",        FloatK)
  , ("for",          ForK)
  , ("if",           IfK)
  , ("goto",         GotoK)
  , ("implements",   ImplementsK)
  , ("import",       ImportK)
  , ("instanceof",   InstanceofK)
  , ("int",          IntK)
  , ("interface",    InterfaceK)
  , ("long",         LongK)
  , ("native",       NativeK)
  , ("new",          NewK)
  , ("package",      PackageK)
  , ("private",      PrivateK)
  , ("protected",    ProtectedK)
  , ("public",       PublicK)
  , ("return",       ReturnK)
  , ("short",        ShortK)
  , ("static",       StaticK)
  , ("strictfp",     StrictfpK)
Java.x

--- remove all underscores from a string
stripUnderscores :: String -> String
stripUnderscores = filter (/=='_')

--- insert an "o" as the second character, so that 'read' interprets as an octal number
precondition: input has at least two characters
insertOctalO :: String -> String
insertOctalO (zero : rest) = zero : 'o' : rest
insertOctalO [] = panic "Zero-character octal literal"

--- convert a binary literal into a number; there is nothing built-in that does this
readBin :: Integral a => String -> a
readBin (_zero : _b : digs) = go 0 (map digitToInt digs)
where
  go acc [b] = acc
  go acc (b : bs) = go (acc * 2 + fromIntegral b) bs
readBin other = panic $ "Zero- or one-character binary literal: " ++ other

--- understand a floating point literal; IRREG: no guarantees that this implements
-- exactly the IEEE standard
lexFloatLiteral :: String -> Token
lexFloatLiteral str = LiteralT $ go1 0 (stripUnderscores str)
where
  go1 :: Integer -> String -> Literal
  go1 acc []                   = DoubleL (fromInteger acc) -- this shouldn’t happen (no suffix)
  go1 acc ('.' : chars)        = go2 (fromInteger acc) 0.1 chars
  go1 acc (e : chars) | is_e e = go3 (fromInteger acc) chars
  go1 acc [f]                 = FloatL (fromInteger acc)
  go1 acc [d]                 = DoubleL (fromInteger acc)
  go1 acc (num : chars)        = go1 (acc * 10 + toInteger (digitToInt num)) chars

  go2 :: Rational   -- what we have so far
  go2 acc []                   = DoubleL (fromRational acc)
  go2 acc _mult [] = DoubleL (fromRational acc)
  go2 acc _mult (e : chars) | is_e e = go3 (fromRational acc) chars
  go2 acc _mult [f] | is_f f = FloatL (fromRational acc)
  go2 acc _mult [d] | is_d d = DoubleL (fromRational acc)
  go2 acc mult (num : chars) = go2 (acc + mult * fromIntegral (digitToInt num))

  go3 :: Rational -- what we have so far
  go3 acc (+' : chars) = go4 acc True 0 chars
  go3 acc (-' : chars) = go4 acc False 0 chars
  go3 acc chars = go4 acc True 0 chars

  go4 :: Rational -- what we have so far
  go4 acc '+' True 0 chars
  go4 acc '-' False 0 chars
  go4 acc chars = go4 acc chars

  go5 :: Rational -- what we have so far
  go5 acc '+' True 0 chars
  go5 acc '-' False 0 chars
  go5 acc chars = go5 acc chars

  go6 :: Rational -- what we have so far
  go6 acc '+' True 0 chars
  go6 acc '-' False 0 chars
  go6 acc chars = go6 acc chars

  go7 :: Rational -- what we have so far
  go7 acc '+' True 0 chars
  go7 acc '-' False 0 chars
  go7 acc chars = go7 acc chars

  go8 :: Rational -- what we have so far
  go8 acc '+' True 0 chars
  go8 acc '-' False 0 chars
  go8 acc chars = go8 acc chars

  go9 :: Rational -- what we have so far
  go9 acc '+' True 0 chars
  go9 acc '-' False 0 chars
  go9 acc chars = go9 acc chars
Java.x

-> Literal
422:  go4 acc pos exp_acc [] = DoubleL (finish acc pos exp_acc)
423:  go4 acc pos exp_acc [f] | is_f f = FloatL (finish acc pos exp_acc)
424:  go4 acc pos exp_acc [d] | is_d d = DoubleL (finish acc pos exp_acc)
425:  go4 acc pos exp_acc (num : chars) = go4 acc pos (exp_acc * 10 + toInteger (digitToInt num)) chars
426:
427:  finish :: Fractional a
428:     => Rational -- base number
429:     -> Bool      -- is the exponent positive?
430:     -> Integer   -- exponent
431:     -> a         -- computed value
432:     finish acc pos exp_acc = fromRational (acc * (10 ^^ maybe_neg pos exp_acc))
433:     -- if the Bool is False, negate the number
434:  maybe_neg :: Bool -> Integer -> Integer
435:  maybe_neg pos num | pos       = num
436:     otherwise = -num
437:
438:  is_f 'f' = True
439:  is_f 'F' = True
440:  is_f _   = False
441:
442:  is_d 'd' = True
443:  is_d 'D' = True
444:  is_d _   = False
445:
446:  is_e 'e' = True
447:  is_e 'E' = True
448:  is_e _   = False
449:
450:  -- accept a separator
451:  sep :: Separator -> AlexAction ()
452:  sep s = tokenS (\_ -> SeparatorT s)
453:
454:  -- accept an operator
455:  op :: Operator -> AlexAction ()
456:  op o = tokenS (\_ -> OperatorT o)
457:
458:
459:  -----------------------------------------------------------------------------
460:  -- The main entry point into this file
461:  -----------------------------------------------------------------------------
462:
463:  -- | Lex some Java code, issuing an error in the Java monad if there is
464:  -- a lexical error.
465:  lexJava :: String -> Java [Token]
466:  lexJava = eitherToJava . lexJavaEither
467:
468:  -- | Lex Java. Returns either an error string (suitable to be reported to
469:  -- a user) or the list of tokens in the file.
470:  lexJavaEither :: String -> Either String [Token]
471:  lexJavaEither input = runAlex input (alexMonadScan >> (fmap reverse alexGetUserState))
472: )