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.
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     = \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?

@Digits @ExponentPart? $FloatTypeSuffix?
@Digits @ExponentPart $FloatTypeSuffix?
@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 = \ \{ \ \{ btnfr""\"\"\" \} \\
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: \\
162: -- String literals, S3.10.5 \\
163: @StringCharacter = $InputCharacter # ["\"\"] \\
164: | @EscapeSequence \\
165: \\
166: @StringLiteral = " @StringCharacter* " \\
167: \\
168: -- This line tells alex that the actual lexing patterns come next. \\
169: -- The word before the ":-" is completely ignored. \\
170: java :- \\
171: \\
172: -- Ignore comments and whitespace \\
173: @Comment ; \\
174: @WhiteSpace ; \\
175: \\
176: -- Identifiers, S3.8 \\
177: $JavaLetter $JavaLetterOrDigit* { tokenS identifier } \\
178: \\
179: -- Integer literals, S3.10.1 \\
180: -- IRREG: Despite the JLS's exhortation, we do *not* fail on overflowing literals. \\
181: @DecimalNumeral { tokenS (LiteralT . IntL . read . stripUnderscores) } \\
182: @DecimalNumeral $IntegerTypeSuffix { tokenS (LiteralT . LongL . read . stripUnderscores . init) } \\
183: \\
184: @HexNumeral { tokenS (LiteralT . IntL . read . stripUnderscores) } \\
185: @HexNumeral $IntegerTypeSuffix { tokenS (LiteralT . LongL . read . stripUnderscores . init) } \\
186: \\
187: @OctalNumeral { tokenS (LiteralT . IntL . read . insertOctalO . stripUnderscores) } \\
188: @OctalNumeral $IntegerTypeSuffix { tokenS (LiteralT . LongL . read . insertOctalO . stripUnderscores . init) } \\
189: \\
190: @BinaryNumeral { tokenS (LiteralT . IntL . readBin . stripUnderscores) } \\
191: @BinaryNumeral $IntegerTypeSuffix { tokenS (LiteralT . LongL . readBin . stripUnderscores . init) } \\
192: \\
193: @DecimalFloatingPointLiteral { tokenS lexFloatLiteral } \\
194: @HexadecimalFloatingPointLiteral { tokenS \(\\sideq \unimplemented \$ "lex hexademical literals like " ++ \s) } \\
195: \\
196: @CharacterLiteral { tokenS (LiteralT . CharL . charHtoJ . read . insertOctalEScapes) } \\
197: \\
198: @StringLiteral { tokenS (LiteralT . StringL . stringHtoJ . read . insertOctalEScapes) } \\
199: \\
200: -- Separators, S3.11 \\
201: "(" { sep LParenS } \\
202: ")" { sep RParenS } \\
203: "(" { sep LBracketS } \\
204: ")" { sep RBracketS } \\
205: \\
206: "[" { sep LBracketS } \\
207: "]" { sep RBracketS }
Operators, S3.12

---

This brace marks the start of Haskell code that will process the above directives

---

This region is all internal Alex plumbing. Haskell experts only!

---

type AlexUserState = [Token] -- in reverse order for easy consing

alexUpdUserState :: ([Token] -> [Token]) -> Alex ()
alexUpdUserState upd = do
  st <- alexGetUserState
  alexSetUserState (upd st)

alexGetString :: AlexInput -> Int -> String
alexGetString (_,_,_,str) len = take len str

tokenS :: (String -> Token) -> AlexInput -> Int -> Alex ()
tokenS f input len = do
  tok = f (alexGetString input len)
  tokenS f input len = do

let tok = f (alexGetString input len)
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]
xInitUserState = []

Utility functions used during lexing
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]
  [ "byte",         ByteK]
  [ "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",     StaticfpK]
351: , ("super", SuperK)
352: , ("switch", SwitchK)
353: , ("synchronized", SynchronizedK)
354: , ("this", ThisK)
355: , ("throw", ThrowK)
356: , ("throws", ThrowsK)
357: , ("transient", TransientK)
358: , ("try", TryK)
359: , ("void", VoidK)
360: , ("volatile", VolatileK)
361: , ("while", WhileK)
362: ]
363:
364: -- remove all underscores from a string
365: stripUnderscores :: String -> String
366: stripUnderscores = filter (/= '_')
367:
368: -- insert an "o" as the second character, so that 'read' interprets as an octal number
369: -- precondition: input has at least two characters
370: insertOctalO :: String -> String
371: insertOctalO (zero : rest) = zero : 'o' : rest
372: insertOctalO [] = panic "Zero-character octal literal"
373:
374: -- convert a binary literal into a number; there is nothing built-in that does this
375: readBin :: Integral a => String -> a
376: readBin (_zero : _b : digs) = go 0 (map digitToInt digs)
377:   where
378:     go :: Integral a => a -> 
379:         [Int] -> a
380:         go acc [] = acc
381:         go acc (b : bs) = go (acc * 2 + fromIntegral b) bs
382:     readBin other = panic $ "Zero- or one-character binary literal: " ++ other
383:
384: -- understand a floating point literal; IRREG: no guarantees that this implements
385: -- exactly the IEEE standard
386: lexFloatLiteral :: String -> Token
387: lexFloatLiteral str = LiteralT $ go1 0 (stripUnderscores str)
388:   where
389:     -- look for the integer part, before the decimal point (if any)
390:     go1 :: Integer -> String -> Literal
391:     go1 acc ('.' : chars) = go2 (fromInteger acc) 0.1 chars
392:     go1 acc (e : chars) | is_e e = go3 (fromInteger acc) chars
393:     go1 acc [f] | is_f f = FloatL (fromInteger acc)
394:     go1 acc [d] | is_d d = DoubleL (fromInteger acc)
395:     go1 acc (num : chars) = go1 (acc * 10 + toInteger (digitToInt num)) chars
396:     -- look for the part after the decimal
397:     go2 :: Rational -- what we have so far
398:         -> Rational -- multiplier for next digit
399:         -> String
400:         -> Literal
401:         -> String
402:         -> Literal
403:         -> Rational
404:         -> String
405:         -> Literal
406:         -> String
407:         -- look for the part after the exponent
408:         go3 :: Rational -- what we have so far
409:         -> String
410:         -> String
411:         -> String
412:         -> String
413:         -> String
414:         -> String
415:         -> String
416:         -- look for the part after the exponent sign
417:         go4 :: Rational -- what we have so far
418:         -> String
419:         -> String
420:         -> String
421:         -> String
421: -> 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:
434: -- if the Bool is False, negate the number
435: maybe_neg :: Bool -> Integer -> Integer
436: maybe_neg pos num | pos       = num
437: | otherwise = -num
438:
439: is_f 'f' = True
440: is_f 'F' = True
441: is_f _  = False
442:
443: is_d 'd' = True
444: is_d 'D' = True
445: is_d _  = False
446:
447: is_e 'e' = True
448: is_e 'E' = True
449: is_e _  = False
450:
451: -- accept a separator
452: sep :: Separator -> AlexAction ()
453: sep s = tokenS (
        _ -> SeparatorT s)
454:
455: -- accept an operator
456: op :: Operator -> AlexAction ()
457: op o = tokenS (_ -> OperatorT o)
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))