```
using System;
using System.IO;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Globalization;

/// <summary>
/// Name: Markus Johan Haugsdal
/// Class: CSC 446 Compiler Construction
/// Assignment: 1
/// Due Date: 01.02.2017
/// Instructor: Hamer
///
/// Description: Lexical scanner for determining tokens
///
/// </summary>

namespace Assignment1
{
    public class lexicalScanner
    {

        //variables
        char ch;
        public static int i = 0;
        //public static string token;
        string lexeme;
        private string fileName;
        private StreamReader sr;
        public static bool hasDot = false;


        public enum SYMBOL { begint, modult, constt, proct, ist,
            ift, thent, elset, elseift, whilet, loopt,
            floatt, integert, chart, gett, putt, endt, ort,
            remt, modt, andt, eoft, unkownt,
            relopt, addopt, assignopt, multopt, lparent,
            rparent,commat,colont,semicolont,
            periodt,idt,literalt,numt };

        //Token object for token building
        public class Token
        {
            public SYMBOL token = SYMBOL.unkownt;
            public string lexeme;
            public int value;
            public float valueR;
            public string literal;

        }



        //Dictionary for checking reserved word tokens
```



```
        Dictionary<string, SYMBOL> reswords =
            new Dictionary<string, SYMBOL>(StringComparer.OrdinalIgnoreCase);

        public lexicalScanner(string fileName)
        {
            this.fileName = fileName;
        }

        public lexicalScanner(string fileName, StreamReader sr) : this
          (fileName)
        {
            this.sr = sr;
        }

        //Gets next char and iterates position in file by 1
        public char getNextChar()
        {

            ch =(char)sr.Read();
            return ch;
        }
        //Gets next char but does NOT iterate position
        public char peekNextChar()
        {
            return (char)sr.Peek();
        }

        /// <summary>
        /// Checks first char and starts building the token
        /// </summary>
        /// <returns> Object of type "token" </returns>
        public Token getNextToken()
        {

            Token token = new Token();

            while (!sr.EndOfStream)
            {
                //Only peek!
                ch = peekNextChar();
                if (!Char.IsWhiteSpace(ch) && Enum.IsDefined
                    (typeof(SYMBOL), SYMBOL.unkownt))
                {
                    //If peek was successful, get next char
                    ch = getNextChar();
                    processToken(token); //Process the token

                    return token;

                }
                //If newline or whitespace
                if (ch == 10 || Char.IsWhiteSpace(ch))
                {
                    ch = getNextChar();
                }
                else
```

```
112                 {
113                     return token;
114                 }
115             }//end while eof
116              //Final token to signify that eof was reached. (Maybe not
                   necessary)
117             token.token = SYMBOL.eoft;
118             return token;
119         }
120         /// <summary>
121         /// ProcessToken.
122         /// Creates the lexeme. Checks first position
123         /// of lexeme to determine what to do.
124         /// </summary>
125         /// <param name="token"></param>
126         public void processToken(Token token)
127         {
128             lexeme = ch.ToString();
129 
130             ch = peekNextChar();
131             //peek?
132 
133             if (Char.IsLetter(lexeme[0])) //IF LETTER
134             {
135 
136                 processWordToken( token);
137 
138             }
139             else if (Char.IsDigit(lexeme[0])) //IF NUMBER
140             {
141                 processNumToken(token);
142             }
143             else if (lexeme[0] == 45) //IF DOUBLE MINUS (COMMENT)
144             {
145                 if (ch == 45)//processComment
146                 {
147                     sr.ReadLine();
148 
149                 }
150                 else
151                 {
152                     processSingleToken(token);
153                 }
154             }
155 
156             //IF single and or double
157             else if (lexeme[0] == 60 || lexeme[0] == 62 ||
158                      lexeme[0] == 61 || lexeme[0] == 47 ||
159                      lexeme[0] == 58 || lexeme[0] == 40 ||
160                      lexeme[0] == 41 || lexeme[0] == 44 ||
161                      lexeme[0] == 59 || lexeme[0] == 34 ||
162                      lexeme[0] == 46   )
163 
164             {
165                 //check next token for =
166                 if (ch == 61)
```



```
167                 {
168                     //process double token
169                     processDoubleToken(token);
170 
171                 }
172                 else
173                 {
174                     //Process Single token
175                     processSingleToken(token);
176                 }
177             }
178 
179             else
180             {
181                 token.token = SYMBOL.unkownt;
182                 token.lexeme = lexeme;
183             }
184             // return lexeme;
185 
186 
187         }
188 
189 
190         /// <summary>
191         /// Processes double tokens.
192         /// </summary>
193         /// <param name="token"></param>
194         private void processDoubleToken(Token token)
195         {
196 
197             lexeme = lexeme + ch.ToString();
198             token.lexeme = lexeme;
199             getNextChar();
200 
201             if (lexeme[0] == 47 || lexeme[0] == 60 || lexeme[0] == 62)
202             {
203                 token.token = SYMBOL.relopt;
204             }
205             else if(lexeme[0] == 58)
206             {
207                 token.token = SYMBOL.assignopt;
208             }
209 
210         }
211         /// <summary>
212         /// Single tokens.
213         /// </summary>
214         /// <param name="token"></param>
215         private void processSingleToken(Token token)
216         {
217 
218             //if =, >, <
219 
220             token.lexeme = lexeme;
221             if (lexeme[0] == 61 || lexeme[0] == 60 || lexeme[0] == 62)
222             {
```

```
223                 token.token = SYMBOL.relopt;
224
225             }
226             else if (lexeme[0] == 43 || lexeme[0] == 45)
227             {
228                 token.token = SYMBOL.addopt;
229
230             }
231             else if(lexeme[0] == 42 || lexeme[0] == 47)
232             {
233                 token.token = SYMBOL.multopt;
234
235             }
236             else if(lexeme[0] == 40)
237             {
238                 token.token = SYMBOL.lparent;
239             }
240             else if (lexeme[0] == 41)
241             {
242                 token.token = SYMBOL.rparent;
243             }
244             else if (lexeme[0] == 44)
245             {
246                 token.token = SYMBOL.commat;
247             }
248             else if (lexeme[0] == 58)
249             {
250                 token.token = SYMBOL.colont;
251             }
252             else if (lexeme[0] == 59)
253             {
254                 token.token = SYMBOL.semicolont;
255             }
256             else if (lexeme[0] == 46)
257             {
258                 token.token = SYMBOL.periodt;
259             }
260             else if (lexeme[0] == 34)
261             {
262                 processStringLiteral(token);
263             }
264
265
266         }
267         /// <summary>
268         /// Processes word tokens. Iterates until it finds illegal character
              ()
269         /// </summary>
270         /// <param name="token"></param>
271         public void processWordToken( Token token)
272         {
273             //Console.WriteLine();
274
275
276             while(sr.Peek() > -1) // read the rest of the lexeme
277             {
```



```
278                 char c = peekNextChar();
279                 //idt can be letters, underscore and/or digits
280                 if (!Char.IsLetterOrDigit(c) && c != 95)
281                     break;
282
283                 else if (sr.Peek() == 32 || sr.Peek() == 10)
284                 {
285                     break;
286                 }
287                 else
288                 {
289                     ch = getNextChar();
290                     lexeme = lexeme + ch;
291                 }
292
293             }//end while
294
295             token.lexeme = lexeme;
296 //            Console.WriteLine("Lexeme: "+lexeme); //GOT IT!
297
298
299             if(reswords.ContainsKey(lexeme))
300             {
301                 //Console.WriteLine("Reserved" );
302                 //If lexeme is reserved word, used reserved token tag
303                 reswords.TryGetValue(lexeme, out token.token);
304                 //Console.WriteLine(token.token);
305             }
306             else
307             {
308
309                 token.token = SYMBOL.idt;
310             }
311             //If lexeme is a reserved word
312
313             if(token.lexeme.Length > 17)
314             {
315                 token.token = SYMBOL.unkownt;
316             }
317
318
319         }
320         /// <summary>
321         /// String literals, if singleToken detects the opening "
322         /// </summary>
323         /// <param name="token"></param>
324         private void processStringLiteral(Token token)
325         {
326             string literal = "\"";
327
328             /*while(sr.Peek() != 34)
329             {
330                 ch = getNextChar();
331             }*/
332             //Look for opening literal
333             //found!
```

```
334             //literal = ch.ToString();
335
336             //ch = getNextChar(); // set ch to '"'
337             getNextChar();
338
339
340             while (sr.Peek() != 34 && sr.Peek() != 10)
341             {
342
343                 literal = literal + ch;
344                 getNextChar();
345
346
347             }
348
349             if (sr.Peek() != 10)
350             {
351                 literal = literal + ch;
352                 ch = getNextChar();
353                 literal = literal + ch;
354                 token.token = SYMBOL.literalt;
355                 token.literal = literal;
356                 token.lexeme = literal;
357             }
358             else
359             {
360
361                 token.token = SYMBOL.unkownt;
362
363                 token.literal = literal;
364                 token.lexeme = literal;
365             }
366
367         }
368
369         /// <summary>
370         /// Num tokens. Iterates and looks for the . which signifies float/
              real
371         /// </summary>
372         /// <param name="token"></param>
373         public void processNumToken(Token token)
374         {
375            // getNextChar();
376             hasDot = false;
377             string nums = lexeme[0].ToString();
378             token.token = SYMBOL.numt;
379             char p;
380             if (char.IsWhiteSpace(ch)) // only one char
381             {
382                 token.lexeme = nums;
383                 Int32.TryParse(nums, out token.value);
384             }
385             else
386             {
387
388                 //char p = peekNextChar(); // peek at next position
```



```
389                 while (sr.Peek() > -1)
390                 {
391                     p = peekNextChar();
392                     if (char.IsDigit(p) || p == 46)
393                     {
394                         getNextChar();
395
396
397                         nums = nums + ch;
398                         if (char.IsWhiteSpace(p))
399                         {
400                             break;
401                         }
402                         if (ch == 46 || p == 46)
403                         {
404                             hasDot = true;
405                         }
406
407                     }
408                     else
409                     {
410                         //Check for
411                         //token.token = "unkownt";
412                         break;
413                     }
414
415
416                 }//end while
417
418
419
420                 if (hasDot == true)
421                 {
422                     if (ch == 46) // if last char of number is .
423                     {
424                         token.token = SYMBOL.unkownt;
425                         token.lexeme = nums;
426                     }
427                     else
428                     {
429                         token.lexeme = nums;
430                         token.valueR = float.Parse(nums);
431                     }
432
433                 }
434
435                 else
436                 {
437
438                     //Int32.TryParse(nums, out token.value);
439                     token.lexeme = nums;
440                     token.value = Int32.Parse(nums);
441
442                 }
443
444
```

```
445     }
446
447
448
449
450         }
451
452         /// <summary>
453         /// Dictionary for reserved words
454         /// </summary>
455         public void createDictionary()
456         {
457             reswords.Add("begin", SYMBOL.begint);
458             reswords.Add("module", SYMBOL.modult);
459             reswords.Add("constant", SYMBOL.constt);
460             reswords.Add("procedure", SYMBOL.proct);
461             reswords.Add("is", SYMBOL.ist);
462             reswords.Add("if", SYMBOL.ift);
463             reswords.Add("then", SYMBOL.thent);
464             reswords.Add("else", SYMBOL.elset);
465             reswords.Add("elseif", SYMBOL.elseift);
466             reswords.Add("while", SYMBOL.whilet);
467             reswords.Add("loop", SYMBOL.loopt);
468             reswords.Add("float", SYMBOL.floatt);
469             reswords.Add("integer", SYMBOL.integert);
470             reswords.Add("char", SYMBOL.chart);
471             reswords.Add("get", SYMBOL.gett);
472             reswords.Add("put", SYMBOL.putt);
473             reswords.Add("end", SYMBOL.endt);
474             reswords.Add("or", SYMBOL.ort);
475             reswords.Add("rem", SYMBOL.remt);
476             reswords.Add("mod", SYMBOL.modt);
477             reswords.Add("and", SYMBOL.andt);
478
479
480         }
481
482         /// <summary>
483         /// PrintToken method to show all tokens and its attribute
484         /// </summary>
485         /// <param name="token"></param>
486         public void printToken(Token token)
487         {
488             string output = "";
489
490
491             if (!String.IsNullOrEmpty(token.lexeme))
492             {
493
494                 i++;
495                 if (token.token == SYMBOL.numt)
496                 {
497                     if (lexicalScanner.hasDot == false)
498                     {
499                         output = string.Format("{0,-15}  {1,-15}",
                        token.token, token.value);
```



```
500
501                     }
502
503                     else
504
505                         output = string.Format("{0,-15}  {1,-15} ",
                        token.token, token.valueR);
506                 }
507                 else if (token.token == SYMBOL.literalt)
508                 {
509
510                     output = string.Format("{0,-15}  {1,-15} ", token.token,
                    token.lexeme);
511
512                 }
513                 else
514                 {
515                      output = string.Format("{0,-15}  {1,-15}", token.token ,
                    token.lexeme);
516                 }
517
518                 Console.WriteLine(output);
519             }
520
521         } // end print token
522
523     }
524 }
525
```