

myname1.c (Page 1 of 1)

```
1: /*-----*/
2: /* myname1.c */
3: /* Author: Bob Dondero */
4: /*-----*/
5:
6: #include <stdio.h>
7: #include <stdlib.h>
8:
9: /* Read a name from stdin, somehow use it, and write it to stdout.
10:  Return 0 if successful, and EXIT_FAILURE otherwise. */
11:
12: int main(void)
13: {
14:     enum {ARRAY_LENGTH = 100};
15:     char acName[ARRAY_LENGTH];
16:     char *pcRet;
17:
18:     printf("What is your name?\n");
19:     pcRet = fgets(acName, ARRAY_LENGTH, stdin);
20:     if (pcRet == NULL)
21:     {
22:         fprintf(stderr, "Read failure.\n");
23:         return EXIT_FAILURE;
24:     }
25:
26:     /* Do something with acName. */
27:
28:     printf("Thank you %s", acName);
29:     return 0;
30: }
31:
32: /*-----*/
33:
34: /* Sample executions:
35:
36: $ gcc217 myname1.c -o myname1
37:
38: $ ./myname1
39: What is your name?
40: Bob
41: Thank you Bob
42:
43: $ ./myname1
44: What is your name?
45: Bob
46: Thank you Bob
47:
48: $ ./myname1
49: What is your name?
50: Bob
51: Thank you Bob
52:
53: */
54:
```

myname2.c (Page 1 of 1)

```
1: /*-----*/
2: /* myname2.c */
3: /* Author: Bob Dondero */
4: /*-----*/
5:
6: #include <stdio.h>
7: #include <stdlib.h>
8:
9: /* Accept a name as argv[1], somehow use it, and write it to stdout.
10:  Return 0 if successful, and EXIT_FAILURE otherwise. */
11:
12: int main(int argc, char *argv[])
13: {
14:     char *pcName;
15:
16:     if (argc != 2)
17:     {
18:         fprintf(stderr, "Wrong number of command-line arguments.\n");
19:         return EXIT_FAILURE;
20:     }
21:
22:     pcName = argv[1];
23:
24:     /* Do something with pcName. */
25:
26:     printf("Thank you %s\n", pcName);
27:     return 0;
28: }
29:
30: /*-----*/
31:
32: /* Sample executions:
33:
34: $ gcc217 myname2.c -o myname2
35:
36: $ ./myname2 Bob
37: Thank you Bob
38:
39: $ ./myname2 Bob
40: Thank you Bob
41:
42: $ ./myname2 Bob
43: Thank you Bob
44:
45: */
```

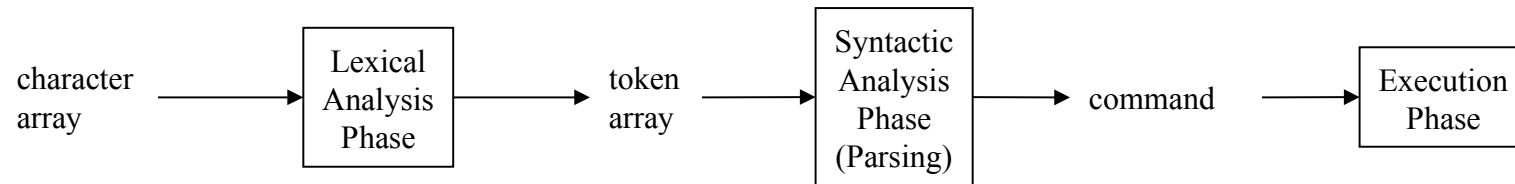
myname3.c (Page 1 of 1)

```
1: /*-----*/
2: /* myname3.c */
3: /* Author: Bob Dondero */
4: /*-----*/
5:
6: #include <stdio.h>
7: #include <stdlib.h>
8:
9: /* Accept a name as an environment variable, somehow use it, and
10:  write it to stdout. Return 0 if successful, and EXIT_FAILURE
11:  otherwise. */
12:
13: int main(int argc, char *argv[])
14: {
15:     char *pcName;
16:
17:     pcName = getenv("MYNAME");
18:     if (pcName == NULL)
19:     {
20:         fprintf(stderr, "No MYNAME environment variable.\n");
21:         return EXIT_FAILURE;
22:     }
23:
24:     /* Do something with pcName. */
25:
26:     printf("Thank you %s\n", pcName);
27:     return 0;
28: }
29:
30: /*-----*/
31:
32: /* Sample executions:
33:
34: $ gcc217 myname3.c -o myname3
35:
36: $ export MYNAME=Bob
37:
38: $ ./myname3
39: Thank you Bob
40:
41: $ ./myname3
42: Thank you Bob
43:
44: $ ./myname3
45: Thank you Bob
46:
47: */
48:
```

commands_demo (Page 1 of 1)

```
1: echo "*** EXTERNAL COMMANDS"
2: /bin/date
3: date
4: pwd
5: ls
6: ls -al
7: echo
8: echo one two three
9: echo one "two three" four
10: printenv
11: printenv PATH
12: printenv HOME
13:
14: echo "*** SHELL BUILTIN COMMANDS (cd, setenv, unsetenv)"
15: cd /usr
16: pwd
17: cd bin
18: pwd
19: cd
20: pwd
21: setenv XXX 123
22: printenv
23: printenv XXX
24: unsetenv XXX
25: printenv
26: printenv XXX
27:
28: echo "*** REDIRECTION"
29: pwd > junk
30: cat junk
31: cat < junk
32: cat < junk > junk2
33: cat junk2
34: cat > junk3 < junk2
35: cat junk3
36: cat junk2 > junk4 junk3
37: cat junk4
38: rm junk junk2 junk3 junk4
39:
40: echo "*** ERRONEOUS COMMANDS"
41: echo "one"
42: cd dir1 dir2
43: setenv
44: unsetenv
45: echo one >
46: echo one > junk1 > junk2
47: cat < nosuchfile
48: nosuchcommand
49:
50: echo "*** UNUSUAL COMMANDS"
51: echo one ">" junk
52: echo one > ">"
53: cat ">"
54: rm ">"
```

Princeton University
 COS 217: Introduction to Programming Systems
 Ish Design



echo one two three > myfile	echo one two three > myfile	Command name: echo Command arg: one Command arg: two Command arg: three Command stdin: NULL Command stdout: myfile
-----------------------------	--	---

echo one "two three" > myfile	echo one two three > myfile	Command name: echo Command arg: one Command arg: two three Command stdin: NULL Command stdout: myfile
-------------------------------	---	---

echo one two > myfile three	echo one two > myfile three	Command name: echo Command arg: one Command arg: two Command arg: three Command stdin: NULL Command stdout: myfile
-----------------------------	--	---

Princeton University

COS 217: Introduction to Programming Systems

The Shell Assignment: Development Stages

Stage 0: Preliminaries

Learn the overall structure of **ish** and the pertinent background information.

Study the assignment specification and the assignment supplement. Review the lecture slides and precept material from the first half of the course on testing, building, debugging, style, *and especially modularity*. Study the lecture slides and precept material from the second half of the course on exceptions and processes, process management, I/O management, signals, and alarms. Complete the pertinent required reading, especially Chapter 8 of *Computer Systems: A Programmer's Perspective* (Bryant & O'Hallaron).

Stage 1: Lexical Analysis

Compose a lexical analyzer module whose input is a sequence of characters from a **character array** and whose output is a **token array**.

Compose a top-level client named **ishlex.c**. Use **ishlex.c**, your lexical analyzer module, and any additional modules that you have composed to build a program named **ishlex**. **ishlex** must read a line from **stdin**, write the line to **stdout**, pass the line to your lexical analyzer module, accept the token array that your lexical analyzer module generates, write the tokens to **stdout**, and repeat until **EOF** (simulated by Ctrl-d).

Test your **ishlex** program (and thus your lexical analyzer module) thoroughly by comparing its behavior with that of the given **sampleishlex** program.

Stage 2: Syntactic Analysis (alias Parsing)

Compose a syntactic analyzer module whose input is a **token array** and whose output is a **command**.

Compose a top-level client named **ishsyn.c**. Use **ishsyn.c**, your lexical analyzer module, your syntactic analyzer module, and any additional modules that you have composed to build a program named **ishsyn**. **ishsyn** must read a line from **stdin**, write the line to **stdout**, pass the line to your lexical analyzer module, accept the token array that your lexical analyzer module generates, pass the token array to your syntactic analyzer module, accept the command that your syntactic analyzer module generates, write the command to **stdout**, and repeat until **EOF** (simulated by Ctrl-d).

Test your **ishsyn** program (and thus your syntactic analyzer module) thoroughly by comparing its behavior with that of the given **sampleishsyn** program.

Stage 3: External Commands

Compose a top-level client named **ish.c**. Use **ish.c**, your lexical analyzer module, your syntactic analyzer module, and any additional modules that you have composed to build a program named **ish**. Your **ish** should execute simple external commands. That is, your **ish** should assume that neither **stdin** nor **stdout** is redirected. Use the **fork()**, **execvp()**, and **wait()** system-level functions.

Your **ish** must read a line from **stdin**, write the line to **stdout**, pass the line to your lexical analyzer module, accept the token array that your lexical analyzer module generates, pass the token array to your syntactic analyzer module, accept the command that your syntactic analyzer module generates, execute the command, and repeat until **EOF** (simulated by Ctrl-d).

Test your **ish** program thoroughly by comparing its behavior when executing simple external commands with that of the given **sampleish** program.

Stage 4: Shell Built-In Commands

Enhance your **ish** program so it can execute the shell built-in commands **exit**, **cd**, **setenv**, **unsetenv**.

Test your **ish** program thoroughly by comparing its behavior when handling shell built-in commands with that of the given **sampleish** program. Specifically, test the program's handling of the **cd** shell built-in command by executing it and the **pwd** and **ls** external commands. Test the program's handling of the **setenv** and **unsetenv** shell built-in commands by executing them and the **printenv** external command. Test the program's handling of the **exit** shell built-in command by executing it.

Stage 5: I/O Redirection

Enhance your **ish** program so it can execute external commands that redirect **stdin** and/or **stdout**. Use the **creat()**, **open()**, **close()**, and **dup()** or **dup2()** system-level functions.

Test your **ish** program thoroughly by comparing its behavior when handling external commands that contain redirection with that of the given **sampleish** program.

Stage 6: Signal Handling

The challenge part.

Copyright © 2018 by Robert M. Dondero, Jr.

dfa.c (Page 1 of 7)

```

1: /*-----*/
2: /* dfa.c */
3: /* Author: Bob Dondero */
4: /*-----*/
5:
6: #include "dynarray.h"
7: #include <ctype.h>
8: #include <stdio.h>
9: #include <stdlib.h>
10: #include <string.h>
11: #include <assert.h>
12:
13: /*-----*/
14:
15: /* The name of the executable binary file. */
16: static const char *pcPgmName;
17:
18: /*-----*/
19:
20: /* A Token object can be either a number or a word. */
21: enum TokenType {TOKEN_NUMBER, TOKEN_WORD};
22:
23: /*-----*/
24:
25: /* A Token is either a number or a word, expressed as a string. */
26:
27: struct Token
28: {
29:     /* The type of the token. */
30:     enum TokenType eType;
31:
32:     /* The string which is the token's value. */
33:     char *pcValue;
34: };
35:
36: /*-----*/
37:
38: /* If no lines remain in psFile, then return NULL. Otherwise read a line
39:    of psFile and return it as a string. The string does not contain a
40:    terminating newline character. The caller owns the string. */
41:
42: static char *readLine(FILE *psFile)
43: {
44:     const size_t INITIAL_LINE_LENGTH = 2;
45:     const size_t GROWTH_FACTOR = 2;
46:
47:     size_t uLineLength = 0;
48:     size_t uPhysLineLength = INITIAL_LINE_LENGTH;
49:     char *pcLine;
50:     int iChar;
51:
52:     assert(psFile != NULL);
53:
54:     /* If no lines remain, return NULL. */
55:     if (feof(psFile))
56:         return NULL;
57:     iChar = fgetc(psFile);
58:     if (iChar == EOF)
59:         return NULL;
60:
61:     /* Allocate memory for the string. */
62:     pcLine = (char*)malloc(uPhysLineLength);
63:     if (pcLine == NULL)

```

dfa.c (Page 2 of 7)

```

64:     {perror(pcPgmName); exit(EXIT_FAILURE);}
65:
66:     /* Read characters into the string. */
67:     while ((iChar != '\n') && (iChar != EOF))
68:     {
69:         if (uLineLength == uPhysLineLength)
70:         {
71:             uPhysLineLength *= GROWTH_FACTOR;
72:             pcLine = (char*)realloc(pcLine, uPhysLineLength);
73:             if (pcLine == NULL)
74:                 {perror(pcPgmName); exit(EXIT_FAILURE);}
75:         }
76:         pcLine[uLineLength] = (char)iChar;
77:         uLineLength++;
78:         iChar = fgetc(psFile);
79:     }
80:
81:     /* Append a null character to the string. */
82:     if (uLineLength == uPhysLineLength)
83:     {
84:         uPhysLineLength++;
85:         pcLine = (char*)realloc(pcLine, uPhysLineLength);
86:         if (pcLine == NULL)
87:             {perror(pcPgmName); exit(EXIT_FAILURE);}
88:     }
89:     pcLine[uLineLength] = '\0';
90:
91:     return pcLine;
92: }
93:
94: /*-----*/
95:
96: /* Write all tokens in oTokens to stdout. First write the number
97: tokens; then write the word tokens. */
98:
99: static void writeTokens(DynArray_T oTokens)
100: {
101:     size_t u;
102:     size_t uLength;
103:     struct Token *psToken;
104:
105:     assert(oTokens != NULL);
106:
107:     uLength = DynArray_getLength(oTokens);
108:
109:     printf("Numbers: ");
110:     for (u = 0; u < uLength; u++)
111:     {
112:         psToken = DynArray_get(oTokens, u);
113:         if (psToken->eType == TOKEN_NUMBER)
114:             printf("%s ", psToken->pcValue);
115:     }
116:     printf("\n");
117:
118:     printf("Words: ");
119:     for (u = 0; u < uLength; u++)
120:     {
121:         psToken = DynArray_get(oTokens, u);
122:         if (psToken->eType == TOKEN_WORD)
123:             printf("%s ", psToken->pcValue);
124:     }
125:     printf("\n");
126: }

```

dfa.c (Page 3 of 7)

```

127:
128: /*-----*/
129:
130: /* Free all of the tokens in oTokens. */
131:
132: static void freeTokens(DynArray_T oTokens)
133: {
134:     size_t u;
135:     size_t uLength;
136:     struct Token *psToken;
137:
138:     assert(oTokens != NULL);
139:
140:     uLength = DynArray_getLength(oTokens);
141:
142:     for (u = 0; u < uLength; u++)
143:     {
144:         psToken = DynArray_get(oTokens, u);
145:         free(psToken->pcValue);
146:         free(psToken);
147:     }
148: }
149:
150: /*-----*/
151:
152: /* Create and return a token whose type is eTokenType and whose
153:    value consists of string pcValue. The caller owns the token. */
154:
155: static struct Token *newToken(enum TokenType eTokenType,
156:     char *pcValue)
157: {
158:     struct Token *psToken;
159:
160:     assert(pcValue != NULL);
161:
162:     psToken = (struct Token*)malloc(sizeof(struct Token));
163:     if (psToken == NULL)
164:         {perror(pcPgmName); exit(EXIT_FAILURE);}
165:     psToken->eType = eTokenType;
166:     psToken->pcValue = (char*)malloc(strlen(pcValue) + 1);
167:     if (psToken->pcValue == NULL)
168:         {perror(pcPgmName); exit(EXIT_FAILURE);}
169:     strcpy(psToken->pcValue, pcValue);
170:
171:     return psToken;
172: }
173:
174: /*-----*/
175:
176: /* Lexically analyze string pcLine. If pcLine contains a lexical
177:    error, then return NULL. Otherwise return a DynArray object
178:    containing the tokens in pcLine. The caller owns the DynArray
179:    object and the tokens that it contains. */
180:
181: static DynArray_T lexLine(const char *pcLine)
182: {
183:     /* lexLine() uses a DFA approach. It "reads" its characters from
184:        pcLine. The DFA has these three states: */
185:     enum LexState {STATE_START, STATE_IN_NUMBER, STATE_IN_WORD};
186:
187:     /* The current state of the DFA. */
188:     enum LexState eState = STATE_START;
189:

```

dfa.c (Page 4 of 7)

```

190:  /* An index into pcLine. */
191:  size_t uLineIndex = 0;
192:
193:  /* Pointer to a buffer in which the characters comprising each
194:     token are accumulated. */
195:  char *pcBuffer;
196:
197:  /* An index into the buffer. */
198:  int uBufferIndex = 0;
199:
200:  char c;
201:  struct Token *psToken;
202:  DynArray_T oTokens;
203:  int iSuccessful;
204:
205:  assert(pcLine != NULL);
206:
207:  /* Create an empty token DynArray object. */
208:  oTokens = DynArray_new(0);
209:  if (oTokens == NULL)
210:      {perror(pcPgmName); exit(EXIT_FAILURE);}
211:
212:  /* Allocate memory for a buffer that is large enough to store the
213:     largest token that might appear within pcLine. */
214:  pcBuffer = (char*)malloc(strlen(pcLine) + 1);
215:  if (pcBuffer == NULL)
216:      {perror(pcPgmName); exit(EXIT_FAILURE);}
217:
218:  for (;;)
219:  {
220:      /* "Read" the next character from pcLine. */
221:      c = pcLine[uLineIndex++];
222:
223:      switch (eState)
224:      {
225:          /* Handle the START state. */
226:          case STATE_START:
227:              if (c == '\0')
228:              {
229:                  free(pcBuffer);
230:                  return oTokens;
231:              }
232:              else if (isdigit(c))
233:              {
234:                  pcBuffer[uBufferIndex++] = c;
235:                  eState = STATE_IN_NUMBER;
236:              }
237:              else if (isalpha(c))
238:              {
239:                  pcBuffer[uBufferIndex++] = c;
240:                  eState = STATE_IN_WORD;
241:              }
242:              else if (isspace(c))
243:                  eState = STATE_START;
244:              else
245:              {
246:                  fprintf(stderr, "Invalid line\n");
247:                  free(pcBuffer);
248:                  freeTokens(oTokens);
249:                  DynArray_free(oTokens);
250:                  return NULL;
251:              }
252:              break;

```

dfa.c (Page 5 of 7)

```

253:
254:     /* Handle the IN_NUMBER state. */
255:     case STATE_IN_NUMBER:
256:         if (c == '\0')
257:         {
258:             /* Create a NUMBER token. */
259:             pBuffer[uBufferIndex] = '\0';
260:             psToken = newToken(TOKEN_NUMBER, pBuffer);
261:             iSuccessful = DynArray_add(oTokens, psToken);
262:             if (! iSuccessful)
263:                 {perror(pcPgmName); exit(EXIT_FAILURE);}
264:             uBufferIndex = 0;
265:             free(pBuffer);
266:             return oTokens;
267:         }
268:         else if (isdigit(c))
269:         {
270:             pBuffer[uBufferIndex++] = c;
271:             eState = STATE_IN_NUMBER;
272:         }
273:         else if (isspace(c))
274:         {
275:             /* Create a NUMBER token. */
276:             pBuffer[uBufferIndex] = '\0';
277:             psToken = newToken(TOKEN_NUMBER, pBuffer);
278:             iSuccessful = DynArray_add(oTokens, psToken);
279:             if (! iSuccessful)
280:                 {perror(pcPgmName); exit(EXIT_FAILURE);}
281:             uBufferIndex = 0;
282:             eState = STATE_START;
283:         }
284:         else
285:         {
286:             fprintf(stderr, "Invalid line\n");
287:             free(pBuffer);
288:             freeTokens(oTokens);
289:             DynArray_free(oTokens);
290:             return NULL;
291:         }
292:         break;
293:
294:     /* Handle the IN_WORD state. */
295:     case STATE_IN_WORD:
296:         if (c == '\0')
297:         {
298:             /* Create a WORD token. */
299:             pBuffer[uBufferIndex] = '\0';
300:             psToken = newToken(TOKEN_WORD, pBuffer);
301:             iSuccessful = DynArray_add(oTokens, psToken);
302:             if (! iSuccessful)
303:                 {perror(pcPgmName); exit(EXIT_FAILURE);}
304:             uBufferIndex = 0;
305:             free(pBuffer);
306:             return oTokens;
307:         }
308:         else if (isalpha(c))
309:         {
310:             pBuffer[uBufferIndex++] = c;
311:             eState = STATE_IN_WORD;
312:         }
313:         else if (isspace(c))
314:         {
315:             /* Create a WORD token. */

```

dfa.c (Page 6 of 7)

```

316:         pcBuffer[uBufferIndex] = '\0';
317:         psToken = newToken(TOKEN_WORD, pcBuffer);
318:         iSuccessful = DynArray_add(oTokens, psToken);
319:         if (! iSuccessful)
320:             {perror(pcPgmName); exit(EXIT_FAILURE);}
321:         uBufferIndex = 0;
322:
323:         eState = STATE_START;
324:     }
325:     else
326:     {
327:         fprintf(stderr, "Invalid line\n");
328:         free(pcBuffer);
329:         freeTokens(oTokens);
330:         DynArray_free(oTokens);
331:         return NULL;
332:     }
333:     break;
334:
335:     default:
336:         assert(0);
337: }
338: }
339: }
340:
341: /*-----*/
342:
343: /* A "number" consists of decimal digit characters, and a "word"
344:    consists of alphabetic characters. Read a line from stdin. Write
345:    the line to stdout. If the line contains a non-number or a non-word,
346:    then write an error message to stderr and reject the line.
347:    Otherwise write to stdout each word that the line contains
348:    followed by each word that the line contains. Repeat until EOF.
349:    Return 0 iff successful. As always, argc is the command-line
350:    argument count and argv is an array of command-line arguments. */
351:
352: int main(int argc, char *argv[])
353: {
354:     char *pcLine;
355:     DynArray_T oTokens;
356:     int iRet;
357:
358:     pcPgmName = argv[0];
359:
360:     printf("-----\n");
361:
362:     while ((pcLine = readLine(stdin)) != NULL)
363:     {
364:         printf("Line: %s\n", pcLine);
365:         iRet = fflush(stdout);
366:         if (iRet == EOF)
367:             {perror(pcPgmName); exit(EXIT_FAILURE);}
368:         oTokens = lexLine(pcLine);
369:         if (oTokens != NULL)
370:         {
371:             writeTokens(oTokens);
372:             freeTokens(oTokens);
373:             DynArray_free(oTokens);
374:         }
375:
376:         printf("-----\n");
377:         free(pcLine);
378:     }

```

```
379:  
380:     return 0;  
381: }
```