myname1.c (Page 1 of 1)

```
/*--------------------------------------------------------------------*/
/* myname1.c                                                          */
/* Author: Bob Dondero                                                */
/*--------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>

/* Read a name from stdin, somehow use it, and write it to stdout.
   Return 0 if successful, and EXIT_FAILURE otherwise. */

int main(void)
{
   enum {ARRAY_LENGTH = 100};
   char acName[ARRAY_LENGTH];
   char *pcRet;

   printf("What is your name?\n");
   pcRet = fgets(acName, ARRAY_LENGTH, stdin);
   if (pcRet == NULL)
   {
      fprintf(stderr, "Read failure.\n");
      return EXIT_FAILURE;
   }

   /* Do something with acName. */

   printf("Thank you %s", acName);
   return 0;
}

/*--------------------------------------------------------------------*/

/* Sample executions:

$ gcc217 myname1.c -o myname1

$ ./myname1
What is your name?
Bob
Thank you Bob

$ ./myname1
What is your name?
Bob
Thank you Bob

$ ./myname1
What is your name?
Bob
Thank you Bob

*/

```

myname2.c (Page 1 of 1)

```c
/*--------------------------------------------------------------------*/
/* myname2.c                                                          */
/* Author: Bob Dondero                                                */
/*--------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>

/* Accept a name as argv[1], somehow use it, and write it to stdout.
   Return 0 if successful, and EXIT_FAILURE otherwise. */

int main(int argc, char *argv[])
{
   char *pcName;

   if (argc != 2)
   {
      fprintf(stderr, "Wrong number of command-line arguments.\n");
      return EXIT_FAILURE;
   }

   pcName = argv[1];

   /* Do something with pcName. */

   printf("Thank you %s\n", pcName);
   return 0;
}

/*--------------------------------------------------------------------*/

/* Sample executions:

$ gcc217 myname2.c -o myname2

$ ./myname2 Bob
Thank you Bob

$ ./myname2 Bob
Thank you Bob

$ ./myname2 Bob
Thank you Bob

*/
```

myname3.c (Page 1 of 1)

```
/*--------------------------------------------------------------------*/
/* myname3.c                                                          */
/* Author: Bob Dondero                                                */
/*--------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>

/* Accept a name as an environment variable, somehow use it, and
   write it to stdout. Return 0 if successful, and EXIT_FAILURE
   otherwise. */

int main(int argc, char *argv[])
{
   char *pcName;

   pcName = getenv("MYNAME");
   if (pcName == NULL)
    {
      fprintf(stderr, "No MYNAME environment variable.\n");
      return EXIT_FAILURE;
   }

   /* Do something with pcName. */

   printf("Thank you %s\n", pcName);
   return 0;
}

/*--------------------------------------------------------------------*/

/* Sample executions:

$ gcc217 myname3.c -o myname3

$ export MYNAME=Bob

$ ./myname3
Thank you Bob

$ ./myname3
Thank you Bob

$ ./myname3
Thank you Bob

*/

```

commands_demo (Page 1 of 1)

```
echo "*** EXTERNAL COMMANDS"
/bin/date
date
pwd
ls
ls -al
echo
echo one two three
echo one "two   three" four
printenv
printenv PATH
printenv HOME

echo "*** SHELL BUILTIN COMMANDS (cd, setenv, unsetenv)"
cd /usr
pwd
cd bin
pwd
cd
pwd
setenv XXX 123
printenv
printenv XXX
unsetenv XXX
printenv
printenv XXX

echo "*** REDIRECTION"
pwd > junk
cat junk
cat < junk
cat < junk > junk2
cat junk2
cat > junk3 < junk2
cat junk3
cat junk2 > junk4 junk3
cat junk4
rm junk junk2 junk3 junk4

echo "*** ERRONEOUS COMMANDS"
echo "one
cd dir1 dir2
setenv
unsetenv
echo one >
echo one > junk1 > junk2
cat < nosuchfile
nosuchcommand

echo "*** UNUSUAL COMMANDS"
echo one ">" junk
echo one > ">"
cat ">"
rm ">"
```

Princeton University
COS 217: Introduction to Programming Systems
Ish Design

$$\begin{tikzcd}
\text{character array} \arrow[r] & \boxed{\text{Lexical Analysis Phase}} \arrow[r] & \text{token array} \arrow[r] & \boxed{\text{Syntactic Analysis Phase (Parsing)}} \arrow[r] & \text{command} \arrow[r] & \boxed{\text{Execution Phase}}
\end{tikzcd}$$

`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

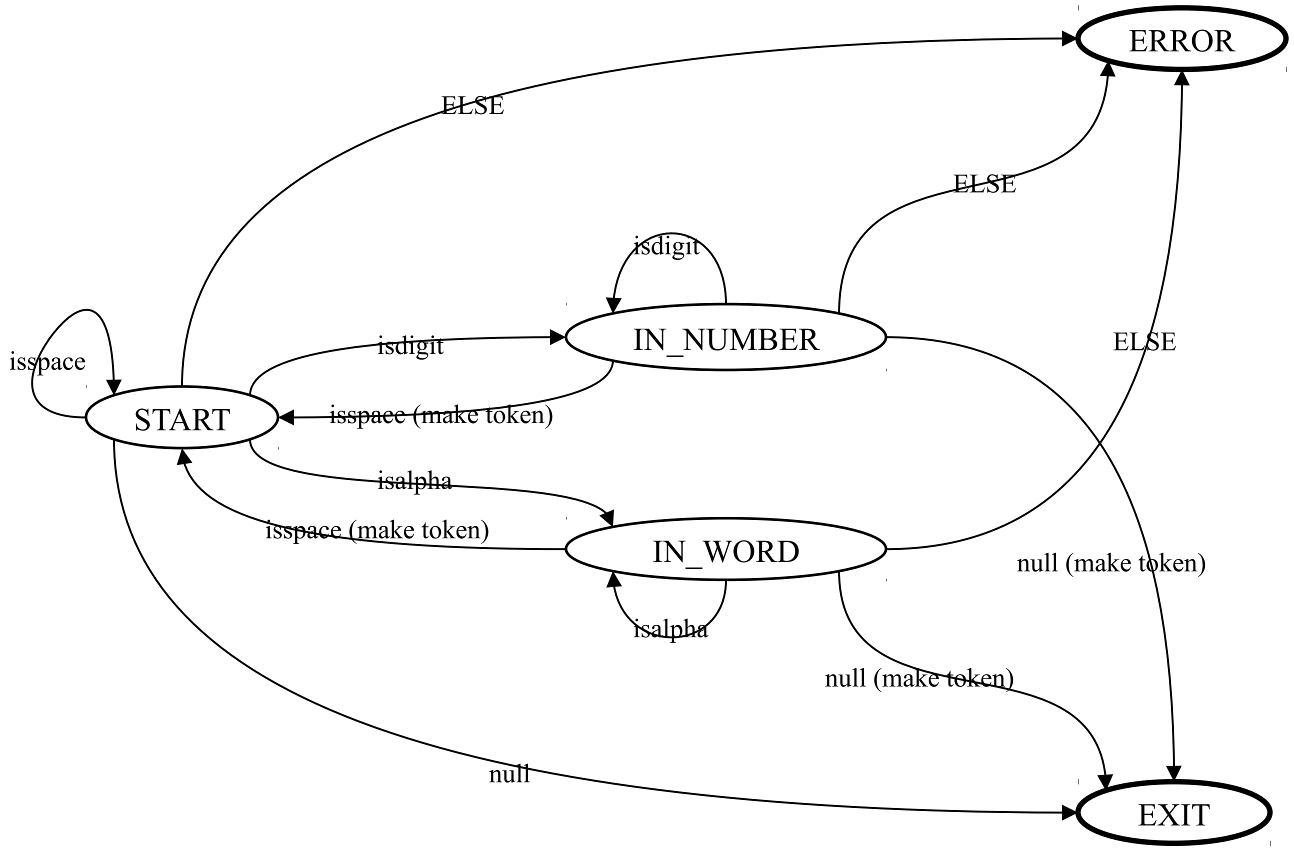
Enhance your **`ish`** program so it can execute external commands that redirect **`stdin`** and/or **`stdout`**. Use the **`creat()`**, **`open()`**, **`close()`**, and **`dup()`** or **`dup2()`** system-level functions.

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.

Princeton University
COS 217: Introduction to Programming Systems
A Deterministic Finite State Automaton



dfa.c (Page 1 of 7)

```c
/*--------------------------------------------------------------------*/
/* dfa.c                                                              */
/* Author: Bob Dondero                                                */
/*--------------------------------------------------------------------*/

#include "dynarray.h"
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

/*--------------------------------------------------------------------*/

/* The name of the executable binary file. */
static const char *pcPgmName;

/*--------------------------------------------------------------------*/

/* A Token object can be either a number or a word. */
enum TokenType {TOKEN_NUMBER, TOKEN_WORD};

/*--------------------------------------------------------------------*/

/* A Token is either a number or a word, expressed as a string. */

struct Token
{
   /* The type of the token. */
   enum TokenType eType;

   /* The string which is the token's value. */
   char *pcValue;
};

/*--------------------------------------------------------------------*/

/* If no lines remain in psFile, then return NULL. Otherwise read a line
   of psFile and return it as a string. The string does not contain a
   terminating newline character. The caller owns the string. */

static char *readLine(FILE *psFile)
{
   const size_t INITIAL_LINE_LENGTH = 2;
   const size_t GROWTH_FACTOR = 2;

   size_t uLineLength = 0;
   size_t uPhysLineLength = INITIAL_LINE_LENGTH;
   char *pcLine;
   int iChar;

   assert(psFile != NULL);

   /* If no lines remain, return NULL. */
   if (feof(psFile))
      return NULL;
   iChar = fgetc(psFile);
   if (iChar == EOF)
      return NULL;

   /* Allocate memory for the string. */
   pcLine = (char*)malloc(uPhysLineLength);
   if (pcLine == NULL)
```

```c
      {perror(pcPgmName); exit(EXIT_FAILURE);}

   /* Read characters into the string. */
   while ((iChar != '\n') && (iChar != EOF))
   {
      if (uLineLength == uPhysLineLength)
      {
         uPhysLineLength *= GROWTH_FACTOR;
         pcLine = (char*)realloc(pcLine, uPhysLineLength);
         if (pcLine == NULL)
            {perror(pcPgmName); exit(EXIT_FAILURE);}
      }
      pcLine[uLineLength] = (char)iChar;
      uLineLength++;
      iChar = fgetc(psFile);
   }

   /* Append a null character to the string. */
   if (uLineLength == uPhysLineLength)
   {
      uPhysLineLength++;
      pcLine = (char*)realloc(pcLine, uPhysLineLength);
      if (pcLine == NULL)
         {perror(pcPgmName); exit(EXIT_FAILURE);}
   }
   pcLine[uLineLength] = '\0';

   return pcLine;
}

/*--------------------------------------------------------------------*/

/* Write all tokens in oTokens to stdout.  First write the number
   tokens; then write the word tokens. */

static void writeTokens(DynArray_T oTokens)
{
   size_t u;
   size_t uLength;
   struct Token *psToken;

   assert(oTokens != NULL);

   uLength = DynArray_getLength(oTokens);

   printf("Numbers:  ");
   for (u = 0; u < uLength; u++)
   {
      psToken = DynArray_get(oTokens, u);
      if (psToken->eType == TOKEN_NUMBER)
         printf("%s ", psToken->pcValue);
   }
   printf("\n");

   printf("Words:  ");
   for (u = 0; u < uLength; u++)
   {
      psToken = DynArray_get(oTokens, u);
      if (psToken->eType == TOKEN_WORD)
         printf("%s ", psToken->pcValue);
   }
   printf("\n");
}
```

```

/*--------------------------------------------------------------------*/

/* Free all of the tokens in oTokens. */

static void freeTokens(DynArray_T oTokens)
{
   size_t u;
   size_t uLength;
   struct Token *psToken;

   assert(oTokens != NULL);

   uLength = DynArray_getLength(oTokens);

   for (u = 0; u < uLength; u++)
   {
      psToken = DynArray_get(oTokens, u);
      free(psToken->pcValue);
      free(psToken);
   }
}

/*--------------------------------------------------------------------*/

/* Create and return a token whose type is eTokenType and whose
   value consists of string pcValue.  The caller owns the token. */

static struct Token *newToken(enum TokenType eTokenType,
   char *pcValue)
{
   struct Token *psToken;

   assert(pcValue != NULL);

   psToken = (struct Token*)malloc(sizeof(struct Token));
   if (psToken == NULL)
      {perror(pcPgmName); exit(EXIT_FAILURE);}
   psToken->eType = eTokenType;
   psToken->pcValue = (char*)malloc(strlen(pcValue) + 1);
   if (psToken->pcValue == NULL)
      {perror(pcPgmName); exit(EXIT_FAILURE);}
   strcpy(psToken->pcValue, pcValue);

   return psToken;
}

/*--------------------------------------------------------------------*/

/* Lexically analyze string pcLine.  If pcLine contains a lexical
   error, then return NULL.  Otherwise return a DynArray object
   containing the tokens in pcLine.  The caller owns the DynArray
   object and the tokens that it contains. */

static DynArray_T lexLine(const char *pcLine)
{
   /* lexLine() uses a DFA approach.  It "reads" its characters from
      pcLine. The DFA has these three states: */
   enum LexState {STATE_START, STATE_IN_NUMBER, STATE_IN_WORD};

   /* The current state of the DFA. */
   enum LexState eState = STATE_START;

```

```c
   /* An index into pcLine. */
   size_t uLineIndex = 0;

   /* Pointer to a buffer in which the characters comprising each
      token are accumulated. */
   char *pcBuffer;

   /* An index into the buffer. */
   int uBufferIndex = 0;

   char c;
   struct Token *psToken;
   DynArray_T oTokens;
   int iSuccessful;

   assert(pcLine != NULL);

   /* Create an empty token DynArray object. */
   oTokens = DynArray_new(0);
   if (oTokens == NULL)
      {perror(pcPgmName); exit(EXIT_FAILURE);}

   /* Allocate memory for a buffer that is large enough to store the
      largest token that might appear within pcLine. */
   pcBuffer = (char*)malloc(strlen(pcLine) + 1);
   if (pcBuffer == NULL)
      {perror(pcPgmName); exit(EXIT_FAILURE);}

   for (;;)
   {
      /* "Read" the next character from pcLine. */
      c = pcLine[uLineIndex++];

      switch (eState)
      {
         /* Handle the START state. */
         case STATE_START:
            if (c == '\0')
            {
               free(pcBuffer);
               return oTokens;
            }
            else if (isdigit(c))
            {
               pcBuffer[uBufferIndex++] = c;
               eState = STATE_IN_NUMBER;
            }
            else if (isalpha(c))
            {
               pcBuffer[uBufferIndex++] = c;
               eState = STATE_IN_WORD;
            }
            else if (isspace(c))
               eState = STATE_START;
            else
            {
               fprintf(stderr, "Invalid line\n");
               free(pcBuffer);
               freeTokens(oTokens);
               DynArray_free(oTokens);
               return NULL;
            }
            break;
```

dfa.c (Page 5 of 7)

```

         /* Handle the IN_NUMBER state. */
         case STATE_IN_NUMBER:
            if (c == '\0')
            {
               /* Create a NUMBER token. */
               pcBuffer[uBufferIndex] = '\0';
               psToken = newToken(TOKEN_NUMBER, pcBuffer);
               iSuccessful = DynArray_add(oTokens, psToken);
               if (! iSuccessful)
                  {perror(pcPgmName); exit(EXIT_FAILURE);}
               uBufferIndex = 0;
               free(pcBuffer);
               return oTokens;
            }
            else if (isdigit(c))
            {
               pcBuffer[uBufferIndex++] = c;
               eState = STATE_IN_NUMBER;
            }
            else if (isspace(c))
            {
               /* Create a NUMBER token. */
               pcBuffer[uBufferIndex] = '\0';
               psToken = newToken(TOKEN_NUMBER, pcBuffer);
               iSuccessful = DynArray_add(oTokens, psToken);
               if (! iSuccessful)
                  {perror(pcPgmName); exit(EXIT_FAILURE);}
               uBufferIndex = 0;
               eState = STATE_START;
            }
            else
            {
               fprintf(stderr, "Invalid line\n");
               free(pcBuffer);
               freeTokens(oTokens);
               DynArray_free(oTokens);
               return NULL;
            }
            break;

         /* Handle the IN_WORD state. */
         case STATE_IN_WORD:
            if (c == '\0')
            {
               /* Create a WORD token. */
               pcBuffer[uBufferIndex] = '\0';
               psToken = newToken(TOKEN_WORD, pcBuffer);
               iSuccessful = DynArray_add(oTokens, psToken);
               if (! iSuccessful)
                  {perror(pcPgmName); exit(EXIT_FAILURE);}
               uBufferIndex = 0;
               free(pcBuffer);
               return oTokens;
            }
            else if (isalpha(c))
            {
               pcBuffer[uBufferIndex++] = c;
               eState = STATE_IN_WORD;
            }
            else if (isspace(c))
            {
               /* Create a WORD token. */
```

```c
                  pcBuffer[uBufferIndex] = '\0';
                  psToken = newToken(TOKEN_WORD, pcBuffer);
                  iSuccessful = DynArray_add(oTokens, psToken);
                  if (! iSuccessful)
                     {perror(pcPgmName); exit(EXIT_FAILURE);}
                  uBufferIndex = 0;

                  eState = STATE_START;
               }
               else
               {
                  fprintf(stderr, "Invalid line\n");
                  free(pcBuffer);
                  freeTokens(oTokens);
                  DynArray_free(oTokens);
                  return NULL;
               }
               break;

            default:
               assert(0);
         }
      }
}

/*--------------------------------------------------------------------*/

/* A "number" consists of decimal digit characters, and a "word"
   consists of alphabetic characters.  Read a line from stdin.  Write
   the line to stdout. If the line contains a non-number or a non-word,
   then write an error message to stderr and reject the line.
   Otherwise write to stdout each number that the line contains
   followed by each word that the line contains.  Repeat until EOF.
   Return 0 iff successful. As always, argc is the command-line
   argument count and argv is an array of command-line arguments. */

int main(int argc, char *argv[])
{
   char *pcLine;
   DynArray_T oTokens;
   int iRet;

   pcPgmName = argv[0];

   printf("------------------------------------\n");

   while ((pcLine = readLine(stdin)) != NULL)
   {
      printf("Line: %s\n", pcLine);
      iRet = fflush(stdout);
      if (iRet == EOF)
         {perror(pcPgmName); exit(EXIT_FAILURE);}
      oTokens = lexLine(pcLine);
      if (oTokens != NULL)
      {
         writeTokens(oTokens);
         freeTokens(oTokens);
         DynArray_free(oTokens);
      }

      printf("------------------------------------\n");
      free(pcLine);
   }
```

```

    return 0;
}
```