```
/*

  http://bumble.sourceforge.net/books/gh/gh.c

OVERVIEW

  This file is an attempt to create a virtual machine which is particularly
  apt for parsing context free languages. The machine consists of a
  (text) stack, (text) workspace, a "tape" (array) structure, 1 "look-ahead"
  character (called the "peep"), an integer accumulator (for simple counting),
  and 1 flag (true or false). In general all instructions operate on the
  "workspace" buffer.

  The essential idea is a stack/tape combination which can handle the nested
  structures of context-free languages.  It is such a simple idea, that if it
  does work, its hard to think why it hasnt been thought of and implemented
  before.

  The general aim of the machine is to create unix-style stream filters
  which can parse and transform context-free language patterns.

  This file also includes a testing "interpreter" with help information
  and interactive commands for inspecting the machine and program
  state.

PARSING WITH THE MACHINE

  The general procedure for parsing is to 'put' the attribute into the tape,
  the clear the workspace, create a token and push that token onto the stack.
  Then a series of tokens are popped off the stack, compared for sequences,
  and 'reduced' as in a bnf (backus-naur form) grammar. At the same time the
  values or attributes of the tokens are got from the tape and transformed as
  required, and then put back into the tape or printed to stdout.

PARSING CHALLENGES

  simple arithmetic expression parser/compiler
    eg: n = 4*y+(6-x)/2
    actually associativity (or operator precedence)
    is difficult if not impossible for the parse machine.

  parse palindromes.

  toy forth parser compiler
  toy lisp parser/compiler

  token*token* the star comes at the end of the token.

  ideas:
    a machineInfo array to explain each part of the
    machine... stack, peep, tape, accumulator etc

  The syntax for the argument indicates what type of argument it is. Eg: "abc"
  is text; [a-z] in brackets, is a range; [abcd] is a list; [:space:] is a
  character class. This is incorporated into the compile() function.

COMPILING AND RUNNING THIS CODE

  tools:
    bash shell (or any other shell for compiling etc)
    linux (makes things easier), sed, date, vim,
    asciidoctor, enscript - for formatting and printing source code

  * create a printable pdf of the code in 2 columns landscape format
  >> enscript -o - -2r -f Courier7 gh.c | ps2pdf - os.pdf

  All the code is in one file, so compilation should be straight-forward
  I call the executable "pp" for "pattern parser". The file is called
  gh.c because I already had a folder called "pp"

  * compile code
  >> gcc gh.c -o pp

  * a bash alias to run the code
  >> alias pp='cd ~/sf/htdocs/books/gh/; ./pp'

  * a bash function to compile the source code gh.c and add a date stamp
  ---------------------
  ppc() {
     echo "Compiling 'gh.c' as executable 'pp'"
     echo "Datestamp: $(date +%d%b%Y-%I%P)"
     # The line below adds the compile time and date to the version
     cd ~/sf/htdocs/books/gh
     cp gh.c gh.pre.c
     sed -i "/v31415\"; *$/s/v31415/version.$(date +%d%b%Y-%I%P)/" gh.pre.c
     gcc gh.pre.c -o pp
  }
  ,,,

COMPILATION OF SCRIPTS

   In a somewhat reflexive manner the machine will compile
   itself. Or more explicitly the compilation of a script
   happens as follows. The machine loads an assembly-code file
   called "asm.pp". It then uses that script to compile
   a given script into the equivalent assembly-code (and saves
   the code in a text file called something like "script.pp"
   Then it loads that assembly-code file and uses it to process
   the input.

   In order to achieve this I removed line numbers from assembly saving and
   loading, and made jumps relative and allowed labels in assembly scripts.
   This will make hand-coding of assembly code feasible

TO DO

  * write function: commandHelpHtml(...) which will display
    a summary of commands and help in the 'markdown'
    or asciidoctor format, which can then be formatted into
    html5, docbook, latex, or pdf.
  * it will be necesssary to have a way of looping back
    to the start of the syntactical analysis section (shift-reduce
    operation on the stack corresponding to backus-naur grammar rules)
    This is so that all shift-reduction are applied. This in turn
    requires a flag reduce? which is set to true when one reduction
    has occurred, thereby triggering the loop.

  * improve checkInstruction() function which tests for
     appropriate datatypes of parameters (eg jumps must have integer
     jump targets, add cannot have class/list/range datatype parameters...)

  - write a display tape/stack function which
    shows stack elements paired with tape elements.

    - include a negation operator (!) which inverts
      the following test
        eg: ! "tree" { ... }  or "tree"!{d;}
        do something if the workspace is not "tree".
      This can be implemented, I think, by using a
      jumpTrue instead of a jumpFalse instruction immediately after the
      test.

    - implement swap command
    - implement 'substitute' command. (do we need this?)
    - set the input stream to some text which the user enters.
    - implement "r/ <prog>" load a program by searching by name
    - make "r/"  list all programs
```

```
BUGS

    * the scanParameter() function is not really robust. It will
      probably accept parametes like this: abc"def" and just
      ignor the "abc".
    * segmentation fault when growing the program.
    * if labels have trailing space, they dont work.
    * all scripts need an extra space at the end, other wise
      asm.pp cant read the final character.

HISTORY

    I have been working on this off and on for a number of years
    (since about 2003) with many stops and starts in the meantime
    as well as many deadends. In fact I dont even know if the idea
    in itself is sound.

    Nevertheless I have a hunch that this approach to parsing context-free
    languages may be very interesting. We shall see. The code is quite close
    to being useful (2018)

    The idea of a parsing machine derived from thinking about the sed stream
    editor and its limitations. At the very least this virtual machine should
    be able to handle nested structures which "sed" is unable to handle.  So
    lisp-like syntax (+ (* n n) m) should be easily parsable.

    The coding of this version was begun around mid-2014. A number of other
    versions have been written in the past but none was successful or
    complete.


    24 july 2019

     writing parameterFromText() function. This will allow parsing
     multiple parameters to an instruction. The tricky bit is that
     parameterFromText() has to return the last character scanned
     to that the next call to it, will start and the current scan
     position. Once I have multiple parameters, then I can write the
     "replace" command: eg replace "one" "two";

     realised that I need a replace command, and this requires the use of 2
     parameters. Maybe a bit of infrastructure will have to be written. An
     example of the use of "replace" is converting c multiline comments into
     bash style comments.  It would be possible to parse line by line and
     achieve this without "replace" but it is a lot more work.

    23 july 2019

     various bits of tidying up. Still cant accept input from standard
     in for some reason (program hangs and waits for console input)

    22 july 2019

     Implemented the swap instruction (x) to swap current tape cell
     and the workspace buffer.

     fixed a bug in the get command which did not allocate
     enough memory for the stack/workspace buffer.

    20 july 2019

     Its all working more all less!. We can write
       pp -f script.pss input.txt
     and the system compiles the script to assembler, loads it,
     and runs it against the input stream in input.txt. No doubt
     there are many bugs, but the general idea works.

     made progress with asm.pp  class blocks seem to be working.
     some nested blocks work. Asm.pp is at a useful stage. It
     can now compile many scripts. Still need to work out how
     to implement the -f switch (segmentation fault at the moment).
     In theory the process is simple... load asm.pp, run it on
     the script file (-f), then load sav.pp (output of asm.pp) and
     run it on the inputstream.

    19 july 2019

     Bug! when program grows during loading a segmentation fault
     occurs.

     created test.commands.pss which contains simple commands which
     can be parsed and compiled by the asm.pp script.

     Also, realised that the compilation from assembler should stop
     with errors when an undefined instruction is found. Dealt with
     a great many warnings that arise when one uses "gcc -Wall"

     implemented
      command 'cc' adds the input stream character count to the
      workspace buffer
      Also made an automatic newline counter, which is incremented
      every time a \n character is encountered. And the 'll'
      command which appends the newline counter as a string onto the
      end of the workspace buffer.

      Since the main function of this parse-machine is to compile
      "languages" from a text source, the commands above are very
      useful because they allow the compilation script to give
      error messages when the source document is not in the correct
      format (with line number and possibly character count).

      Did some work on "asm.pp" which is the assembler file which
      compiles scripts. Sounds very circular but it works.
      Realised that after applying bnf rules, need to jump back to
      the "parse:" label in case other previous rules apply.

    18 july 2019

      Discovered a bug when running "asm.pp" in unix filter mode
      "Abort trap: 6" which means writing to some memory location
      that I should not be. Strangely, when I run the same script
      interactively (with "rr") it works and doesnt cause the
      abort.

      Created a "write" command, on the machine, which writes the
      current workspace to a file called "sav.pp". This has a parallel
      in sed (which also has a 'w' write command). This command
      should be useful when compiling scripts and then running them
      (since they are compiled to an intermediate "assembler" phase,
      and then loaded into the machine).

      made some progress to use the pattern-machine as a unix-style filter
      program. Added some command line options with getopt().
      The parser should be usable (in the future) like sed: eg
        cat somefile | pp -sf script.pp > result.txt
      or
        cat somefile | pp -sa script.ppa > result.txt
      where script.ppa is an "assembler" listing which can be loaded into
      the machine.

    16 july 2019

      Working on parsing with asm.pp. Seem to have basic commands
      parsing and compiling eg: add "this"; pop; push; etc
      Simple blocks are parsing and compiling.
      There are still some complications concerning the order of
      shift-reductions.
      Made execute() have a return value eg:
        0: success no problems
        1: end of stream reached
        2: undefined instruction
        3: quit/crash executed (exit script)
        4: write command could not open file sav.pp for writing

      more work. some aesthetic fixes to make it easier to see what
      the machine is doing. wrote showMachineTapeProgram() to give a nice
      view of pretty much everything that is going on in the machine at once.
      Working on how to collate "attributes" in the tape array register.
      Made an optional parameter to printSomeTape() that escapes \n \r etc
      in the tape cells which makes the output less messy.

    15 july 2019
      A lot of progress. Starting to work on asm.pp again. Have
      basic shift-reduction of stack tokens working. Now to get
      the tape "compiling" attributes as well.

      The bug seems to be: that JUMP is not treated as a relative
      jump by execute() but is being compiled as a relative jump
      by instructionFromText(). So, either make, JUMPs relative or ...

      Made the "labelTable" (or jumpTable) a property of the program.
      This is a good idea. Also made the command 'jj' print out the
      label table. Still using "jumptable" phrase but this is not
      a good name for this.

      I should organise this file: first structure definitions.
      then prototype declarations, and then functions. I havent done
      this because it was convenient to write the function immediately
      after the structure def (so I could look at the properties).
      But if I rearrange, then it will be easier to put everything in
      a header file, if that is a good idea.

      Lots of minor modifications. made searchHelp also search the
      help command name, for example. Added a compileTime (milliseconds)
      property to the Program structure, and a compileDate (time_t).
      81 instructions (which is how many instructions in asm.pp at the
      moment) are taking 4 milliseconds to compile. which seems pretty
      slow really.
      sizes:
        gh.c 138430 bytes
        pp   80880 bytes

      trying to eliminate warnings from the gcc compiler, which are actually
      very handy. Also seem to have uncovered a bug where the "getJump"
      function was actually after where it was used (and this gh.c does
      not use any header files, which is very primitive). So the
      label jumptable code should not have been working at all...
      changing lots of %d to %ld for long integers. Also, on BSD unix
      the ansi colour escape code for "grey" appears to be black.

    13 july 2019
      Looking at this on an OSX macbook. The code compiles (with a number of
      warnings) and seems to run. The colours in this bash environment are
      different.

    12 Dec 2018
      After stepping through the "asm" program I discovered that
      unconditional jump targets are not being correctly encoded. This
      probably explains why the script was not running properly. Also
      I may put labels into the deassembled listings so that the listings
      are more readable.

    19 sept 2018
      revisiting.
      Need to create command line switches: eg -a <name> for loading
      an assembler script. and -f <name> to load a script file.
      Need to step through the asm.pp script and
      work out why stack reduction is not working... (see above for
      the answer). An infinite
      loop is occurring. Also, need to write the treemap app for iphone
      android, not related to this. Also, need to write a script that
      converts this file and book files to an asciidoctor format for
      publishing in html and pdf. Then send all this to someone more
      knowledgeable.
    5 sept 2018
      gh.c 133423 bytes
      pp   78448 bytes
      Would be handy to have a "run until 10 more chars read" function.
      This would help to debug problematic scripts.

      Segmentation fault probably caused by trying to "put" to
      non-existant tape cell (past the end). Need to check tape size
      before putting, and grow the tape if necessary.

      could try to make a palindrome parser.  Getting a segmentation fault
      when running the asm.pp program right through. Wrote an INTERPRET
      mode for testing- where commands are executed on the machine
      but not compiled into the current program. Wrote runUntilWorkspaceIs()
      and adding a testing command to invoke this. This should make is easier
      to test particular parts of a script.  found and fixed a problem with
      how labels are resolved, this was cause by buildJumpTable() not ignoring
      multiline comments.
    4 sept 2018
      Made multiline comments (#* ... *#) work in assembler scripts.  Made the
      machine.delimiter character visible and used by push and pop in
      execute(). There is no way to set the delimiter char or the escape char
      in scripts
    3 sept 2018
      Added multiline comments to asm.pp (eg #* ... *#) as well
      as single line comments with #.
      Idea: make gh.c produce internal docs in asciidoctor format
      so we can publish to html5/docbook/pdf etc.
      working on the asm.pp script. Made "asm" command reset the
      machine and program and input stream. Added quoted text and
      comments to the asm.pp script parsing, but no stack parsing yet.

      Need to add multiline comments to the loadAssembledProgram() function.
      while and whilenot cannot use a single char:
      eg: whilenot "\n" doesnt work. So, write 'whilenot [\n]' instead
      Also should write checkInstruction() called by
      instructionFromText() to make sure that the instruction has
      the correct parameter types. Eg: add should have parameter type
      text delimited by quotes. Not a list [...] or a range [a-z]

      If the jumptable is a global variable then we can test
      jump calculations interactively. Although its not really
      necessary. Would be good to time how long the machine
      takes to load assembler files, and also how long it takes
      to parse and transform files.
    2 sept 2018
      wrote getJump() and made instructionFromText() lookup the label
      jump table and calculate the relative jump. It appears to be
      working. Which removes perhaps the last obstacle to actually writing
      the script parser. Need to make program listings "page" so I can
      see long listings.
    1 Sept 2018
      writing printJumpTable() and trying to progress. Looking at
      Need to add "struct label table[]" jumptable parameter
      to instructionFromText(), and compile().
      asciidoctor.
    31 aug 2018
      Continued to work on buildJumpTable. Will write printJumpTable.
      Renamed the script assembler to "asm.pp"
      Made a bash function to insert a timestamp. Created an
      "asm" command in the test loop to load the asm.pp file into the program.
      Started a buildTable function for a label jump table. These
      label offsets could be applied by the "compile" function.
    30 August 2018
      "gh.c" source file is 117352 bytes.
      Compiled code is 72800 bytes. I could reduce this dramatically
      by separating the test loop from the machine code.

      Revisiting this after taking a long detour via a forth bytecode
      machine which currently boots on x86 in real mode (see
      http://bumble.sourceforge.net/books/osdev/os.asm ) and then trying
      to port it to the atmega328p architecture (ie arduino) at
      http://bumble.sf.net/books/arduino/os.avr.asm
      The immediate task seems to be to write code to create a label
      table for assembly listings, and then use that code to replace
      labels with relative jump offsets. After that, we can start to write
      the actual code (in asm.pp) which will parse and compile scripts.

      So the process is: the machine loads the script parser code (in
      "asm" format) from a text file. The machine uses that program to
      parse a given script and convert to text "asm" format. The machine
      then loads the new text asm script and uses it to parse and
      transform ("compile") an input text stream.

    20 decembre 2017
      Allowed assembly listings with no line numbers as default. It
      would be good idea to allow labels in assembly listings, eg 'here:' to
      make it easier to hand code assembly. So, need a label table. Look at
      the info arrays for the syntax... Made conditional jumps relative so
      that they would be easier to "hand-code" as integers (although labels
      are really needed).  Also, need to add a loadAsm() function which is
      shorthand to load the script assembler.
    17 decembre 2017
      For some reason, the code was left in a non compilable
      state in 2016. I think the compile() and instructionFromText()
      functions could be rewritten but seem to be working at
      the moment.

   13 dec 2017
     The code is not compiling because the parameter to the "compile()"
     function is wrong. When we display instructions, it would be good to
     always indicate the data type of the parameter (eg: text, int, range etc)
     Modify "test" to use different parameter types, eg list, range, class.

   29 september 2016
     used instructionFromText() within the compile() function and changed
     compile to accept raw instruction text (not command + arguments) wrote
     scanParameter which is a usefull little function to grab an argument up
     to a delimiter. It works out the delimiter by looking at the first char
     of the argument and unescapes all special chars. Now need to change
     loadAssembled to use compile().

   28 sept 2016
     Added a help-search / and a command help search //. Added
     escapeText() and escapeSpecial(), and printEscapedInstruction().
     add writeInstruction() which escapes and writes an instruction
     to file. Added instructionFromText() and a test command
     which tests that function.
     Worked on loadAssembledProgram() to properly load
     formats such as "while [a-z]" and "while [abc\] \\ \r \t]" etc.
     All this work is moving towards having the same parse routine
     loading assembled scripts from text files as well as interactively
     in the test loop.

   26 sept 2016
     Discovered that swap is not implemented.
   22 sept 2016
     Added loadlast, savelast, runzero etc. a few convenience functions
     in the interpreter. One hurdle: I need to be able to write
      testis "\n" etc where \n indicates a newline so that we can
      test for non printing characters. So this needs to go into the
      machine as its ascii code.
    Also, when showing program listings, these special characters
    \n \t \r should be shown in a different colour to make it
    obvious that they are special chars...
    Also: loadprogram is broken at the moment.... need to deal
    with datatypes.

   21 Sept 2016
     When in interpreter mode, reading the last character should not
     exit, it should return to the command prompt for testing purposes.

   15 August 2016
     Wrote an "int read" function which reads one character from
     stdin and simplifies the code greatly. Still need to
     fix "escaping". need to make ss give better output, configurable
     Escaping in 'until' command seems to be working.

   13 August 2016
     Added a couple more interpreter commands to allow the
     manipulation of the program and change the ip pointer. Now
     it is possible to jump the ip pointer to a particular
     instruction. Also, looked at the loadAssembledProgram and
     saveAssembledProgram functions to try to rewrite them correctly.
     The loadAssembledProgram needs to be completely cleaned up
     and the logic revised.
     My current idea is to write a program which transforms a pp
     script into a text assembly format, and then use the
     'loadAssembledProgram' to load that script into the machine.
     Wrote 'runUntilTrue' function which executes
     program instructions until the machine flag is set to true (by
     one of the test instructions, such as testis testbegins, testends...
     This should be useful for debugging complex machine programs.

   7 Jan 2016
     wrote a cursory freeMachine function with supporting functions
   4 Jan 2016
     tidying up the help system. had the idea of a program browser,
     ie browse 'prog' subfolder and load selected program into
     the machine. Need to write the actual script compilation
     code.
   3 Jan 2016
     Writing a compile function which compiles one instruction
     given command and args. changed the cells array in Tape
     to dynamic. Since we can use array subscripts with pointers
     the code hardly changes. Added the testclass test
     Made program.listing and tape.cells pointers with dynamic
     memory allocation.
   1 Jan 2016
     working on compiling function pointers for the character
     class tests with the while and testis instructions.
     Creating reflection arrays for class and testing.

   late Dec 2015
     Continued work. Trying to resolve all "malloc"
     and "realloc" problems. Using a program with instruction listing
     within the machine. Each command executed interactively gets
     added to this.
   26 Dec 2015
     Saving and loading programs as assembler listings.
     validate program function. "until" & "pop" more or less working.
     "testends" working ...

   19 Dec 2015
     Lots of small changes. The code has been polished up to an almost
     useable stage. The machine can be used interactively. Several
     instructions still need to be implemented. Push and pop need to be
     written properly.  Need to realloc() strings when required.  The info
```

```
      structure will include "number of parameter fields" so that the code
      knows how many parameters a given instruction needs. This is useful
      for error checking when compiling.

    16 Dec 2015
      Revisiting this after a break. Got rid of function pointers, and
      individual instruction functions. Just have one executing function
      "execute()" with a big switch statement. Same with the test
      (interpreter) loop. A big switch statement to process user commands.
      Start with the 'read' command.  Small test file. The disadvantage of not
      having individual instruction functions

       (eg void pop(struct Machine * mm)
           void push(struct Machine * mm) etc) is that we cannot
      implement the script compiler as a series of function calls.
      However the "read" instruction does have a dedicated function.

    23 Feb 2015
      The development strategy has been to incrementally add small bits to the
      machine and concurrently add test commands to the interpreter.

    22 Feb 2015
      Had the idea to create a separate test loop file (a command interpreter)
      with a separate help info array. show create showTapeHtml to print the
      tape in html. These functions will allow the code to document itself,
      more or less.

      Changes to make:
      The conditional jumps should be relative, not absolute.  This will make
      it easier to hand write the compiler in "assembly language". Line
      numbers are not necessary in the assembly listings. The unconditional
      jump eg jump 0 can still be an absolute line number.

      Put a field width in the help output.
      Change help output colours. Make "pp" help command "ls"
      make p.x -> px or just "."

   2006 - 2014
      Attempted to write various versions of this machine, in java,
      perl, c++ etc, but none was completed successfully
      see http://bumble.sf.net/pp/cpp for an incomplete implementation
      in c++. But better to look at the current version, which is
      much much better.

   2005 approximately
      I started to think about this parsing machine while living
      in Almetlla de Mar. My initial ideas were prompted by trying
      to write parsing scripts in sed and then reading snippets of
      Compilerbau by N. Wirth, thinking about compilers and grammars


TERMINOLOGY

   * tape pointer:
   * stack: a text buffer that can be manipulated like a stack.
   * command: one of the permitable instructions for
      the VM (eg push pop etc)
   * instruction: a command & parameter (maybe) compiled
      into a 'program' (array) which can be executed by
      the Virtual Machine
   *

*/

// <code>

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <ctype.h>
#include <string.h>
#include <unistd.h>


// not actually used for anything.
#include "gh.h"

/* ------------------------------------------
 stuff that could go in a header file. But I cant be bother moving
 all my structures into a different file.
 function prototypes. But some prototypes must go under the relevant
 structure.
*/
void escapeSpecial(char *, char *);

// ---------------------------


// information about colours, which may be useful since I use
// them alot to display the machine.


enum Bool {TRUE=0, FALSE};


/*
 execute a compiled instruction. Possible return values might be
        0: success no problems
        1: end of stream reached (tried to read eof)
        2: trying to execute undefined instruction
        3: quit/crash command executed (exit script)
        4: could not open 'sav.pp' for writing (from write command)
        5: tried to execute unimplemented command
 */

/* the enum, array of structures, and associated functions are
   to provide readable exit and error codes for functions such as
   execute(), run(), compile(), loadScript(), etc
   EXECQUIT is actually a success code, where as BADQUIT (returned
   by the "bail" command, is an error.
   */
enum ExitCode {
  SUCCESS=0, EXECQUIT, ENDOFSTREAM, EXECUNDEFINED, BADQUIT,
  READSAVERROR, WRITESAVERROR, UNIMPLEMENTED };

struct {
  enum ExitCode error;
  char * description;
} exitCodes[] = {
  { SUCCESS, "success, no errors" },
  { EXECQUIT, "quit was executed (exit script)" },
  { ENDOFSTREAM, "tried to read past end of stream (eof)" },
  { EXECUNDEFINED, "tried to execute undefined machine instruction" },
  { BADQUIT, "program exited with an error" },
  { READSAVERROR, "could not open 'sav.pp' for reading" },
  { WRITESAVERROR, "could not open 'sav.pp' for writing" },
  { UNIMPLEMENTED, "executed an unimplemented machine instruction" }
};

// print the description for an error
void printExitCode(enum ExitCode error) {
  printf("(%d) %s\n", exitCodes[error].error, exitCodes[error].description);
}

enum Colour {
  GREYc=0, REDc, PALEREDc, GREENc, PALEGREENc, BROWNc, YELLOWc,
  PALEBLUEc, BLUEc, PURPLEc, PINKc, AQUAc, CYANc,
  WHITEc, BLACKc, NORMALc
```

```
};

struct {
  enum Colour colour;
  char * name;
  char * ansi;
} colourInfo[] = {
  { GREYc, "grey", "\x1B[1;30m"},
  { REDc, "red",  "\x1B[0;31m"},
  { PALEREDc, "palered", "\x1B[1;31m"},
  { GREENc, "green", "\x1B[0;32m"},
  { PALEGREENc, "palegreen", "\x1B[1;32m"},
  { BROWNc, "brown", "\x1B[0;33m"},
  { YELLOWc, "yellow", "\x1B[1;33m"},
  { PALEBLUEc, "paleblue", "\x1B[0;34m"},
  { BLUEc, "blue", "\x1B[1;34m"},
  { PURPLEc, "purple",  "\x1B[0;35m"},
  { PINKc, "pink", "\x1B[1;35m"},
  { AQUAc, "aqua", "\x1B[0;36m"},
  { CYANc, "cyan", "\x1B[1;36m"},
  { WHITEc, "white", "\x1B[37m"},
  { BLACKc, "black", "\x1B[0;30m"},
  { NORMALc, "normal", "\x1B[0m"}

};

#define BLACK "\x1B[0;30m"

#define PALERED "\x1B[0;31m"
#define RED  "\x1B[1;31m"
#define PALEGREEN  "\x1B[0;32m"
#define GREEN "\x1B[1;32m"
#define BROWN "\x1B[0;33m"
#define YELLOW "\x1B[1;33m"
#define PALEBLUE  "\x1B[0;34m"
#define BLUE "\x1B[1;34m"
#define PALEMAGENTA "\x1B[0;35m"
#define PURPLE "\x1B[0;35m"
#define PINK "\x1B[1;35m"
#define PALECYAN "\x1B[0;36m"
#define AQUA "\x1B[0;36m"
#define CYAN "\x1B[1;36m"
#define WHITE "\x1B[37m"
// on BSD (Mac OSX, this grey seems to be black)
// define GREY "\x1B[1;30m"
#define GREY "\x1B[37m"

#define NORMAL "\x1B[0m"

 //strcpy(GREY, colourInfo[GREYc].name);
 //char *  = colourInfo[c].ansi;
 //char * Colours[] =
  // {RED, GREEN, YELLOW, BLUE, PURPLE, CYAN, WHITE, NORMAL};

 /* show a table of colours and their names and numbers
 */
 void showColours() {
   int ii;
   printf("Colours and colour values: \n");
   for (ii = 0; ii <= NORMALc; ii++) {
     //if (ii % 5 == 0) printf("\n");
     printf("%s%2d: %s%9s [%s%s%s]\n",
       GREEN, ii,
       WHITE, colourInfo[ii].name,
       colourInfo[ii].ansi, colourInfo[ii].name, WHITE);
   }
   printf("\n");
 }
```

```
 /* just displays some text at the bottom of the screen when
    the output needs to be 'paged' (one screen at a time)
 */
 void pagePrompt() {
   printf("\n%s<enter>%s for more (%sq%s = exit):",
     AQUA, NORMAL, AQUA, NORMAL);
 }

 // print text in lots of colours! ....
 void colourPrint(char * text) {
   int ii;
   int length = strlen(text);
   for (ii = 0; ii < length; ii++) {
     printf("%s%c", colourInfo[ii%13 + 1].ansi, *text);
     text++;
   }
   printf(NORMAL);
 }

 //  Print the text as a row in various colours
 void colourRow(char * text) {
   int ii;
   int length = strlen(text);
   for (ii = 0; ii < (40/length); ii++) {
     printf("%s%s", colourInfo[ii%13 + 1].ansi, text);
   }
   printf("\n");
   printf(NORMAL);

 }

 //  print a banner
 void banner() {
   printf("\n");
   colourRow("== ");
   printf("%s pp:%s A Pattern Parsing Machine and Language%s\n",
      PINK, GREEN, NORMAL);
   colourRow("== ");
   printf("\n");
 }

// One cell in the 'tape' or array of strings. The cells of the
// tape have a 1-to-1 correspondance to the cells of the stack
// show capacity be characters or bytes??? for new 'bytes' because
// its easier to program

#define TAPECELLSIZE 50
struct TapeCell {
  int resizings;      // how many mallocs, for performance issues
  size_t capacity;    // how many characters in this cell?
  char * text;        // the text content of the cell
};

// functions relating to TapeCells
// the sizeof(char) below is unnecessary but will be important
// if this is rewritten for wchar_t

/*
 initialise and allocate text memory for a tape cell
 this function could also allocate memory for the TapeCell
 itself, but should it? eg cell = malloc(sizeof(struct TapeCell))
 Well... if the Tape is not dynamically resized, then it is
 not necessary.
   eg: struct Tape { int capacity; struct TapeCell cells[1000]; }
   will allocate the required memory for the tapecells
*/
void newCell(struct TapeCell * cell, int cellSize) {
  cell->text = malloc(cellSize * sizeof(char));
  if(cell->text == NULL) {
    fprintf(stderr,
      "couldnt allocate memory for cell->text in newCell()\n");
    exit(EXIT_FAILURE);
  }
  cell->capacity = cellSize;
  cell->resizings = 0;
}

// not really necessary this func
/*
void freeCell(struct TapeCell * cell) {
  free(cell);
}
*/

// make the tape cell bigger, preserving text (realloc not malloc)
// bug! this function should have a 'minimum' parameter because
// cells contents are set, not added to.
// another bug: the capacity should not be multiplied by
// sizeof char or wchar because cell-capacity should indicate
// how many chars it can hold, not bytes. (but for realloc its ok)
#define TAPECELLGROWFACTOR 50
void growCell(struct TapeCell * cell) {
  long newCapacity =
    cell->capacity + TAPECELLGROWFACTOR*sizeof(char);
  cell->text = realloc(cell->text, newCapacity);
  if(cell->text == NULL) {
    fprintf(stderr,
      "couldnt allocate more memory for cell->text in growCell()\n");
    exit(EXIT_FAILURE);
  }
  cell->capacity = newCapacity;
  cell->resizings++;
}

void printTapeCellInfo(struct TapeCell * tc) {
  printf("%s, %ld ", tc->text, tc->capacity);
}

void printTapeCellLongInfo(struct TapeCell * tc) {
  printf("%s ( capacity=%ld, resizings=%d )",
    tc->text, tc->capacity, tc->resizings);
}

// the tape: should this be a separate structure or
// just an array. For a dynamically resizable tape a separate
// structure seems useful, but adds complexity.

// tape capacity refers to number of tape cells not bytes
#define TAPESTARTSIZE 1000
struct Tape {
  int resizings;  //how many times re-malloced this tape?
  long capacity;  //how many cells in the tape
  struct TapeCell * cells;   // dynamic allocation
  //struct TapeCell cells[TAPESTARTSIZE];   //non dynamic allocation
  int currentCell;
};

// functions relating to tapes

// initialise a tape and initialise each tapecell
// but we dont really need a pointer to a tape to be
// returned from this function, since there will only be
// one tape per machine.
void newTape(struct Tape * tape, int cells, int cellSize) {
  tape->capacity = cells;
  tape->resizings = 0;
  tape->cells = malloc(tape->capacity * sizeof(struct TapeCell));
  if(tape->cells == NULL) {
    fprintf(stderr,
      "couldnt allocate memory for cells in tape in newTape()\n");
    exit(EXIT_FAILURE);
  }
  int ii;
  for (ii = 0; ii < tape->capacity; ii++) {
    newCell(&tape->cells[ii], cellSize);
  }
  tape->currentCell = 0;
}

void freeTape(struct Tape * tape) {
  int ii;
  for (ii = 0; ii < tape->capacity; ii++) {
    free(tape->cells[ii].text);
  }
  free(tape->cells);
}

/*
// if the tape can get more cells, this has to be implemented
struct Tape * growTape() {
}
*/

// set all cells in the tape to empty
void clearTape(struct Tape * tape) {
  int ii;
  tape->currentCell = 0;
  for (ii = 0; ii < tape->capacity; ii++) {
     tape->cells[ii].text[0] = '\0';
  }
}

// show the contents of the tape, cells, with > showing current cell
void printTape(struct Tape * tape) {
  int ii;
  for (ii = 0; ii < tape->capacity; ii++) {
    if (ii == tape->currentCell)
      printf("%4d> %s\n", ii, tape->cells[ii].text);
    else
      printf("%4d: %s\n", ii, tape->cells[ii].text);
  }
}

// show the contents of the tape around current cell
// add a "context" variable, to see more or less of the tape
void printSomeTape(struct Tape * tape, enum Bool escape) {
  int ii;

  int start = tape->currentCell - 3;
  int end = tape->currentCell + 3;
  if (start < 0) start = 0;
  if (end > tape->capacity) end = tape->capacity;

  printf("Tape Size: %ld \n", tape->capacity);
  for (ii = start; ii < end; ii++) {
    if (escape == TRUE) {
      if (ii == tape->currentCell) {
        printf(" %s%3d> [", YELLOW, ii);
        escapeSpecial(tape->cells[ii].text, CYAN);
        printf("%s]%s\n", YELLOW, NORMAL);
      } else {
        printf(" %s%3d%s  [", BLUE, ii, NORMAL);
        escapeSpecial(tape->cells[ii].text, CYAN);
        printf("%s]\n", NORMAL);
      }
    } else {
      if (ii == tape->currentCell) {
        printf(" %s%3d> [%s]%s\n",
          YELLOW, ii, tape->cells[ii].text, NORMAL);
      } else {
        printf(" %s%3d%s  [%s]%s\n",
          BLUE, ii, NORMAL, tape->cells[ii].text, NORMAL);
      }

    }
  }
}

// same as above, but shows cell capacity.
void printSomeTapeInfo(struct Tape * tape, enum Bool escape, int context) {
  int ii;

  // use context
  int start = tape->currentCell - context;
  int end = tape->currentCell + context;
  if (start < 0) { end = end - start; start = 0; }
  if (end > tape->capacity) end = tape->capacity;

  printf("Tape Size: %ld \n", tape->capacity);
  for (ii = start; ii < end; ii++) {
    if (escape == TRUE) {
      if (ii == tape->currentCell) {
        printf(" %s%3ld/%-3ld%s)%s%3d> [",
          BROWN, strlen(tape->cells[ii].text), tape->cells[ii].capacity,
          GREY, YELLOW, ii);
        escapeSpecial(tape->cells[ii].text, CYAN);
        printf("%s]%s\n", YELLOW, NORMAL);
      } else {
        printf(" %s%3ld/%-3ld%s)%s%3d  [",
          BROWN, strlen(tape->cells[ii].text), tape->cells[ii].capacity,
          GREY, BLUE, ii);
        escapeSpecial(tape->cells[ii].text, CYAN);
        printf("%s]\n", NORMAL);
      }
    } else {
      if (ii == tape->currentCell) {
        printf(" %s%3ld/%-3ld%s)%s%3d> [%s]%s\n",
          BROWN, strlen(tape->cells[ii].text), tape->cells[ii].capacity,
          GREY, YELLOW, ii, tape->cells[ii].text, NORMAL);
      } else {
        printf(" %s%3ld/%-3ld%s)%s%3d%s  [%s]\n",
          BROWN, strlen(tape->cells[ii].text), tape->cells[ii].capacity,
          GREY, BLUE, ii, NORMAL, tape->cells[ii].text);
      }

    }
  }
}

// show a detailed display of the tape
void printTapeInfo(struct Tape * tape) {
  int ii;
  char key;
  printf("Tape Size: %ld \n", tape->capacity);
  for (ii = 0; ii < tape->capacity; ii++) {
    if (ii == tape->currentCell)
      printf(" %s%ld%s) %s%3d> [%s]%s\n",
        BROWN, tape->cells[ii].capacity, GREY,
        YELLOW, ii, tape->cells[ii].text, NORMAL);
    else
      printf(" %s%ld%s) %s%3d%s  [%s]\n",
        BROWN, tape->cells[ii].capacity, GREY,
        BLUE, ii, NORMAL, tape->cells[ii].text);
    if ( (ii+1) % 14 == 0 ) {
      pagePrompt();
      key = getc(stdin);
      if (key == 'q') { return; }
    }
  }
}


// The buffer holds the stack and the workspace
// pushing and popping the stack just involves shifting the
// workspace pointer left or right.
struct Buffer {
  int resizings;
  size_t capacity;
  char * workspace;
  char * stack;
};

/* returns the current capacity of the workspace buffer.
This is important during virtual machine instructions such
as "get" and "put" because it allows us to know if we need to
do a malloc etc. */
size_t workspaceCapacity(struct Buffer * buffer) {
  size_t diff = buffer->workspace - buffer->stack;
  return buffer->capacity - diff - 1;
}

#define BUFFERSTARTSIZE 1000
void newBuffer(struct Buffer * buffer, int size) {
  buffer->stack = malloc(size * sizeof(char));
  if(buffer->stack == NULL) {
    fprintf(stderr,
      "couldnt allocate memory for stack/workspace in newBuffer()\n");
    exit(EXIT_FAILURE);
  }
  buffer->workspace = buffer->stack;
  buffer->stack[0] = '\0';
  buffer->resizings = 0;
  buffer->capacity = size - 1;  // one less for \0
}

void clearBuffer(struct Buffer * buffer) {
  buffer->workspace = buffer->stack;
  buffer->stack[0] = '\0';
  buffer->resizings = 0;
}

// make the workspace/stack buffer bigger in the machine
void growBuffer(struct Buffer * buffer, int increase) {
  int offset = buffer->workspace - buffer->stack;
  long newCapacity =
    buffer->capacity + increase + 1*sizeof(char);
  buffer->stack = realloc(buffer->stack, newCapacity);
  if(buffer->stack == NULL) {
    fprintf(stderr,
      "couldnt allocate more memory for buffer in growBuffer()\n");
    exit(EXIT_FAILURE);
  }
  buffer->workspace = buffer->stack + offset;
  buffer->resizings++;
  buffer->capacity = newCapacity - 1;
}

// show the state of the buffer for debugging
void showBufferInfo(struct Buffer * buffer) {
  printf("%scapacity: %s%ld%s \n",
    GREEN, PURPLE, buffer->capacity, NORMAL );
  printf("%sresizings: %s%d%s \n",
    GREEN, PURPLE, buffer->resizings, NORMAL );
    //%.*s
    // not sure how to do this
  //printf("%sstack: %s%.*s%s \n",
  //  GREEN, PURPLE, buffer.workspace - buffer.stack, NORMAL );
  printf("%sworkspace: %s%s%s \n",
    GREEN, PURPLE, buffer->workspace, NORMAL );
  // show capacity, resizings, value of stack, of workspace
  // etc
}

// display buffer
void showBuffer(struct Buffer * buffer) {
  printf("capacity: %ld \n", buffer->capacity);
  // show capacity, resizings, value of stack, of workspace
  // etc
}

// valid commands on the machine. The structures in the info[]
// array should be in the same order for easy access
enum Command {
  // workspace commands
  ADD=0, CLIP, CLOP, CLEAR, REPLACE, PRINT,
  // stack commands
  POP, PUSH,
  // tape commands
  PUT, GET, SWAP, INCREMENT, DECREMENT,
  // read commands
  READ, UNTIL, WHILE, WHILENOT,
  // jumps
  JUMP, JUMPTRUE, JUMPFALSE,
  // tests
  TESTIS, TESTCLASS, TESTBEGINS, TESTENDS, TESTEOF, TESTTAPE,
  // accumulator commands
  COUNT, INCC, DECC, ZERO,
  // append character counter and newline counter
  CHARS, LINES,
  // escape commands
  ESCAPE, UNESCAPE,
  // system commands
  STATE, QUIT, BAIL,
  // write workspace to file
  WRITE,
  NOP,
  // not a recognised command
  UNDEFINED
};

// the following may be usefull commands on the machine

// INDENT, /* indents each line of the workspace */
//   or just make a "sub" string substitution function instead
// REPLACE, or sub
// NEWLINE,/* adds a newline to the  workspace */
// CHECK,   /* a shift reduce jump */
// LABEL,   /* */


// contains help information about all machine commands
struct {
  enum Command c;
  char * name;
  char abbreviation;
  int args;     // how many parameters for this command
  char * shortDesc;
  char * longDesc;
  char * example;
} info[] = {
  { ADD, "add", 'a', 1,
     "adds a given text to the workspace buffer",
     "appends text to the workspace buffer.",
     ":'tree' { put; clear; add 'noun*'; push; }" },
```

```
  { CLIP, "clip", 'k', 0,
     "removes one character from the end of the workspace",
     "One character is removed from the end of the workspace \n"
     "register. The peek register and all other registers are unchanged. \n"
     "This instruction is useful for obtaining the value of a token when \n"
     "the end delimiter of the token is not wanted. ",
     "'\"' { clear; until '\"'; clip; print; } clear; \n "
     " prints all text within quotes but first removing the \n"
     " the quotes. The above example could also have been \n"
     " done with whilenot '\"' and without the clip but in cases \n"
     " where the end delimiter is more than one character, until \n"
     " seems to be required." },
  { CLOP, "clop", 'K', 0,
     "removes one character from the beginning of the workspace",
     "One character is deleted from the beginning of the workspace \n"
     "register. All other registers are unchanged. ",
     "d; add 'abc'; clop; print; \n"
     "prints 'bc' many times. " },
  { CLEAR, "clear", 'd', 0,
     "clears the workspace",
     "This instruction deletes the contents of the workspace \n"
     "register. All other registers are unchanged. The parsing machine \n "
     "is designed so that all text manipulation should take place in \n"
     "the workspace register (which is like a text 'accumulator' or an \n"
     "AX/EAX register in the x86 architecture. The clear instruction is \n"
     "one of a set of pp instructions which modifies the workspace. \n" ,
     "clear;" },
  { REPLACE, "replace", 'D', 0,
     "replace one string with another in the workspace",
     ".... \n" ,
     "replace;" },
  { PRINT, "print", 't', 0,
     "prints the workspace to stdout",
     "prints the content of the workspace register to standard-out. \n"
     "Unlike the sed stream editor, the pattern parser has no implicit \n"
     "print instruction compiled by default. ",
     "t;t;t;d;  \n"
     "   print every character in the input stream 3 times" },
  { POP, "pop", 'p', 0,
    "pops from the stack to the start of the workspace",
    "Pop scans the content of the stack starting from its end  \n"
    "and skipping the '*' delimiter character if it is the \n"
    "last char on the stack \n"
    "and scanning backwards until it encounters \n"
    "another * delimiter character \n"
    "(or whatever is the stack token delimiter). It then adjusts the \n "
    "workspace register pointer so that it points either to the next '*' \n"
    "on the stack, or else to the beginning of the stack. Pop also \n"
    "decrements the tape pointer by one (or does nothing if the \n"
    "tape pointer == 0). If the stack is empty when 'pop' is executed \n"
    "then the state of the machine is unchanged (including the tape \n"
    "pointer)",
    "pop; \n"
    "if the stack == 'adj*noun*' before pop is executed and \n"
    " the workspace == 'verb*fullstop*' then, after the pop \n"
    " the stack will be 'adj*' and the workspace will be \n"
    " noun*verb*fullstop* and the tapepointer will be one less \n"
    },
  { PUSH, "push", 'P', 0,
    "push (*) delimited token from start of the workspace to the stack",
    "Push appends a delimited token from the beginning of the \n"
    "workspace register to the end of the stack register. If there is \n"
    "no text in the workspace then no action is taken and the machine \n"
    "state is unchanged. The amount \n"
    "of text appended is determined by the placement of the token \n"
    "delimiter character (usually *). Push also increments the current \n"
    "tape cell by one, but only if the workspace is not empty. \n",
    "push; \n"
    "  If the workspace was adj*noun*verb* before the push, then after \n"
    "  the push it will be 'noun*verb*'. If the stack was sentence* \n"
    "  before the push, then it will be 'sentence*adj*' after the push. \n"
    "  It should be noted that the 'stack' is really just a character \n"
    "  buffer which can be manipulated in stack-like ways with push and pop" },
  { PUT, "put", 'G', 0,
    "puts the workspace into the current tape cell",
    "copies the content of the machine workspace register \n"
    "into the tape cell currently indicated by the tape pointer. \n"
    "The previous contents of this tape cell are overwritten. \n"
    "The contents of the workspace register are unchanged by this \n"
    "instruction.",
    "put;" },
  { GET, "get", 'g', 0,
    "Appends current tape cell to the workspace",
    "This appends the contents of the current tape cell to the workspace \n"
    "register. The contents of the tape cell are unchanged by this \n"
    "instruction.",
    "put; get; print; clear; \n"
    "  prints every character in the input stream twice.\n"
    "  [note that there is an implicit 'read' at the beginning \n"
    "   of every script, and an implicit 'jump 0' looping instruction \n"
    "   at the end of every script. This is because the pattern parsing \n"
    "   machine is designed to operate on streams, one character at a \n"
    "   time]. " },
  { SWAP, "swap", 'x', 0,
    "swaps the current tape cell and the workspace",
    "The contents of the workspace register and the current \n"
    "tape cell are swapped.",
    "swap; print; clear; \n"
    "   prints each pair of characters in the input stream in \n"
    "   reverse order. That is, if the input stream is 'abcdef' \n"
    "   then the script will print 'badcfe'" },
  { INCREMENT, "++", '>', 0,
    "increments the current tape cell by one",
    "When a push operation on the stack is performed, auto ++",
    "++;" },
  { DECREMENT, "--", '<', 0,
    "decrements the current tape cell by one",
    "The tape cell pointer is decreased by 1. If the tape cell pointer \n"
    "is already zero, then this instruction does nothing. This instruction \n"
    "is useful to access the values/ attributes of parsing tokens on the \n"
    "stack, but the script writer needs to realign the tape pointer, so \n"
    "that subsequent pushes and pops will work correctly. In general, this \n"
    "means that ++ instructions should be matched by -- instructions and \n"
    "vice-versa. ",
    "get; --; get; ++; \n"
    "  gets last 2 attributes from the tape \n"
    "  ... \n" },
  { READ, "read", 'r', 0,
    "read a character from the input stream",
    "The character in the peep register is appended to the end of the \n"
    "workspace register, and then the next character from the input stream \n"
    "is placed in the peep register. If the peep register already holds \n"
    "the end-of-file token, then the program will normally exit. \n\n"
    "All scripts have an implicit 'read' instruction at the very beginning \n"
    "of the program. (The internal compiler places a 'read' instruction \n"
    "at position 0). This is because the pattern parser is designed to \n"
    "cycle through input streams one character at a time. A program which \n"
    "is hand-assembled in the interpreter or a text file does not need to \n"
    "include the initial 'read' instruction, but the program may not be \n"
    "very useful without it." ,
    "t;r;d; \n"
    "  deletes every second character from the input stream \n"
    "  This script would be compiled to 'assembly' as: \n"
    "     read \n"
    "     print \n"
    "     read \n"
    "     clear \n"
    "     jump 0 \n" },
  { UNTIL, "until", 'R', 1,
    "reads the input stream until the workspace ends with given text",
    "While the workspace buffer does not end with the given \n"
    "text, the input stream is read through the peep register and \n"
    "appended to the workspace. However, the 'until' command \n"
    "takes note of the character in the escape register and ignores \n"
    "text ending with characters escaped with that character. \n\n"
    "This instruction is useful for parsing tokens which have a \n"
    "multiple-character end delimiter, such as, html comments which \n"
    "end with -->. Or c comments ending in */ ",
    "[:space:] {d;} '/' { r; '*' {until '*/'; d; } t;d;} t;d;  \n"
    "  deletes c style comments from the input stream \n"
    "  [this script is untested. Also, note that unlike sed \n"
    "  the steam editor in pp there is no implicit 'print' instruction] "},
  { WHILE, "while", 'w', 1,
    "reads io while peek is something",
    "While continues to read the input stream while \n"
    "the character in the 'peek' register is in a particular \n"
    "character class (eg, alphanumeric, integer, whitespace etc) \n"
    "character range (eg p-z) or a character list (eg: 678abc^&*). \n\n"
    "This instruction is useful for parsing tokens which are defined \n"
    "by a set of legal characters, for example c identifiers \n",
    "while :alnum:; w a-f; while abxy; w \"" },
  { WHILENOT, "whilenot", 'W', 1,
    "reads io while peek is not something",
    "The whilenot machine instruction is the complementary \n"
    "instruction to the while instruction. It reads the input stream \n"
    "while the character in the peek register is not a specified \n"
    "character class, character range or character list. \n"
    "",
    "W :punct: \n"
    "  keep reading the input stream and append to the workspace \n"
    "  register as long as the peek register is not a punctuation \n"
    "  character. " },
  { JUMP, "jump", ',', 1,
    "unconditional jump to absolute instruction number",
    "Transfer program control to the given instruction number. \n"
    "This command is automatically compiled into a program as the last \n"
    "instruction in the form JUMP 0. This is because, pp, like sed has \n"
    "an implicit character reading loop. The command is not designed to \n"
    "be used in a script, but can be used in an assembler code text file. ",
    "jump 0; " },
  { JUMPTRUE, "jumptrue", 'j', 1,
    "jump to relative instruction +/-N if flag is set to true ",
    "Jumps if the flag is set to true. This flag is set by \n"
    "instructions such as testis, testbegins etc. The parameter to \n"
    "the instruction is a relative offset of the next \n"
    "instruction to be executed by the machine. \n"
    "This is designed only to be used in assembler code text files. \n"
    "",
    "jumptrue 10  \n"
    "  jumps forward 10 instructions if the flag register is set to true " },
  { JUMPFALSE, "jumpfalse", 'J', 1,
    "jump to relative instruction +/-N if flag is set to false ",
    "If the flag register is set to false, transfer program control \n"
    "to the specified instruction at offset. If the flag register is true \n"
    "the state of the machine is unchanged and the next instruction in the \n"
    "program will be executed. \n\n"
    "This instruction is used in conjuction with the 'test' instructions \n"
    "to implement condition behaviour. In particular, it allows certain \n"
    "instructions to be skipped if certain conditions in the workspace \n"
    "register are met. This is a relative jump",
    "J -10 \n"
    "  jumps back 10 instructions if the flag register is set to false" },
  { TESTIS, "testis", '=', 1,
    "Test if the workspace equals the given text",
    "If the workspace buffer is exactly the same as the text \n"
    "then set the machine flag to true.  \n",
    " \"abc\" { pop; } \n "
    "  pop one item off the machine stack if the workspace \n"
    "  is currently equal to 'abc'." },
  { TESTCLASS, "testclass", '?', 1,
    "test if all characters in the workspace are in given class/list/range",
    "[note: on reflection, this instruction could be amalgamated \n"
    "with the testis instruction. The only difference is the datatype \n"
    "of the parameter- range a-z, list abcd, class :alnum:, text blah etc] \n"
    "This command allows \n"
    "character class tests eg [:space:] tests if \n"
    "the workspace consists entirely of space characters \n"
    "[:digit:] tests that the workspace only contains numeric digits \n"
    "Also range tests, eg: '[a-z]' workspace only has characters "
    "Also list tests, eg: '[abxy]' workspace must contain only the "
    "listed characters ",
    "[:alnum:] { add 'X' } \n "
    "  adds X to the workspace if the workspace consists only of \n"
    "  alphanumeric characters." },
  { TESTBEGINS, "testbegins", 'b', 1,
    "tests if the workspace begins with the given text",
    "If the workspace buffer ends with the specified text "
    "then the flag register is set to false, otherwise it is "
    "set to true. This command is used in conjuction with the "
    "jump instructions to control program flow.",
    ".\"abc\" { clear; } \n"
    "  deletes the content of the workspace if it begins with \n"
    "  'abc' " },
  { TESTENDS, "testends", 'B', 1,
    "tests if the workspace ends with the given text",
    "Check if the workspace register ends with the given text \n"
    "and set the machine flag register to true if so, and to false \n"
    "otherwise. ",
    ",\"abc\" { clear; } \n"
    "  deletes the content of the workspace if it ends with \n"
    "  'abc' " },
  { TESTEOF, "testeof", 'E', 0,
    "tests if the peep register contains the end of file character",
    "long desc...",
    "EOF { print 'end of file'; } \n"
    "   prints 'end of file' when the end of the input stream has \n"
    "   been reached." },
  { TESTTAPE, "testtape", '*', 0,
    "tests if the workspace is the same as the content of the current cell",
    "If the current cell (the cell pointed to by the tape pointer) \n"
    "of the machine tape structure is exactly equal \n"
    "to the content of the workspace register, then the flag register is \n"
    "set to true, otherwise the flag is set to false.",
    "** { print 'equal'; } \n"
    "  print the word equal to standard-out if the tape cell is \n"
    "  the same as the workspace." },
  { COUNT, "count", 'n', 0,
    "appends the value of the accumulator to the workspace",
    "The (long?) integer accumulator is appended as text to the \n"
    "workspace buffer with no preceding space. ",
    "add 'chars read'; count; " },
  { INCC, "a+", '+', 0,
    "increments the machine accumulator by one",
    "increments the integer accumulator in the parsing machine \n"
    "Hopefully this accumulator will be useful in implementing address \n"
    "calculation when 'assembling' programs. This type of calculation \n"
    "is required when converting if and loop blocks in highish level \n"
    "languages to forward jumps in assembler type languages. \n"
    "Since the pattern parsing machine is designed to transform, translate \n"
    "compile and assemble, this is an important function.",
    "a+ ; EOF { add 'total chars = '; count; print; } \n"
    "  for every character read from the input stream, increment \n"
    "  the machine accumulator. At the end of the input stream, \n"
    "  display the total number of characters in the stream/ file. " },
  { DECC, "a-", '-', 0,
    "decrements the accumulator by one",
    "Decrements the integer accumulator in the parsing machine. \n"
    "incc and decc could be used, for example to ensure that opening \n "
    "and closing braces in some programming language are 'balanced'. ",
    "a-; \n"
    "  decrement the accumulator by one." },
  { ZERO, "zero", '0', 0,
    "sets the numerical accumulator to zero",
    "resets the machine accumulator back to zero. ",
    "incc; print; 0; print; " },
  { CHARS, "cc", 'c', 0,
    "Appends number of characters read to end of workspace buffer",
    "Appends number of characters read to end of workspace buffer",
    "add 'chars read='; cc; print; clear; " },
  { LINES, "ll", 'l', 0,
    "Appends number of lines read to end of workspace buffer",
    "Appends number of lines read to end of workspace buffer",
    "add 'lines read='; ll; print; clear; " },
  { ESCAPE, "escape", '^', 1,
    "prefixes the given character with the escape character ",
    "This command replaces a character with itself preceded by \n"
    "the machines escape char (usually \\). This is important in \n"
    "many parsing situations because it allows, for example, a double \n"
    "quote character to be included within double quotes.\n"
    "[note: I need to think through the tricky issues of escaping the \n"
    " escape character itself].",
    "esc ';'; \n"
    "   replace all occurrences of the semi-colon in the workspace \n"
    "   with \\; "},
  { UNESCAPE, "unescape", 'v', 1,
    "converts chars to unescaped equivalent",
    "removes the escape char prefix from the given character ",
    "unescape x;" },
  { STATE, "state", 'S', 0,
    "prints the current state of the machine.",
    "state prints to stdout the values of the registers of "
    "the parsing machine, such as the workspace, the stack, the peep "
    "the tape and the numerical accumulator",
    "state; " },
  { QUIT, "quit", 'q', 0,
    "immediately exits the script",
    "This command ceases all commands and exits the parsing machine \n"
    "process.",
    "quit; " },
  { BAIL, "bail", 'Q', 0,
    "immediately exits the script with a non zero error code",
    "This command performs the same as the 'quit' command, but \n"
    "tells the machine interpreter to generate a non-zero error code \n"
    "on exit.",
    "bail; " },
  { WRITE, "write", 's', 0,
    "writes the workspace to file 'sav.pp'",
    "The contents of the workspace are saved to the file 'sav.pp'  \n"
    "This is also used to convert a script to assembler and then run it",
    "write; " },
  { NOP, "nop", 'o', 0,
    "no operation",
    "this command performs no operation. The program ip is incremented \n"
    "by 1 and the state of the machine is otherwise unchanged. \n"
    "I am not actually convinced \n"
    "that it is necessary, but since most real machines have a nop \n"
    "I will kowtow to the tyranny of consensus.",
    "nop; \n"
    "  the state of the machine is unchanged except that the \n"
    "  instruction pointer is incremented by 1." },
  { UNDEFINED, "undefined", 'e', 0,
    "undefined command",
    "The undefined command is used as a 'bookend' programmatically "
    "and to catch all invalid commands. When a program is cleared "
    "all instruction commands are set to undefined.",
    "n/a" }
};


// commands to test and analyse the machine
// these enumerations are in the same order as the informational
// array below for convenience
enum TestCommand {
  HELP=0, COMMANDHELP, SEARCHHELP, SEARCHCOMMAND, LISTMACHINECOMMANDS,
  DESCRIBEMACHINECOMMANDS, LISTCLASSES, LISTCOLOURS,
  MACHINEPROGRAM, MACHINESTATE, MACHINETAPESTATE, MACHINEMETA, BUFFERSTATE,
  STACKSTATE, TAPESTATE, TAPEINFO, TAPECONTEXTLESS, TAPECONTEXTMORE,
  RESETINPUT, RESETMACHINE,
  STEPMODE, PROGRAMMODE, MACHINEMODE, COMPILEMODE,
  IPCOMPILEMODE, ENDCOMPILEMODE, INTERPRETMODE, LISTPROGRAM, LISTSOMEPROGRAM,
  LISTPROGRAMWITHLABELS, PROGRAMMETA, SAVEPROGRAM,
  SHOWJUMPTABLE, LOADPROGRAM, LOADASM,
  LOADLAST, LOADSAVED, LISTSAVFILE, SAVELAST, CHECKPROGRAM,
  CLEARPROGRAM, CLEARLAST, INSERTINSTRUCTION,
  EXECUTEINSTRUCTION, PARSEINSTRUCTION, TESTWRITEINSTRUCTION,
  STEPCODE, RUNCODE, RUNZERO, RUNTOTRUE, RUNTOWORK, RUNTOENDSWITH,
  IPZERO, IPEND, IPGO, IPPLUS, IPMINUS,
  SHOWSTREAM,
  EXIT, UNKNOWN
};

// stepcode and executeinstruction below seem to be the
// same exactly
struct {
  enum TestCommand c;
  char * names[2];
  char * argText;   // eg <command>
  char * description;
} testInfo[] = {
  { HELP, {"hh", ""}, "",
     "list all interactive commands" },
  { COMMANDHELP, {"H", ""}, "<command>",
     "show help for a given machine command" },
  { SEARCHHELP, {"/", "h/"}, "<search term>",
     "searches help system for an interpreter command containing search term" },
  { SEARCHCOMMAND, {"//", "//"}, "<command search term>",
     "searches help for a machine command containing search term" },
  { LISTMACHINECOMMANDS, {"com", ""}, "",
     "list all machine commands" },
  { DESCRIBEMACHINECOMMANDS, {"Com", ""}, "",
     "list and describe machine commands" },
  { LISTCLASSES, {"class", "cl"}, "",
     "list all valid character classes for testclass and while" },
  { LISTCOLOURS, {"col", "colours"}, "",
     "list all ansi colours" },
  { MACHINEPROGRAM, {"m", ""}, "",
     "show state of machine, tape and current program instructions" },
  { MACHINESTATE, {"M", ""}, "",
     "show the state of the machine" },
  { MACHINETAPESTATE, {"s", ""}, "",
     "show state of the machine buffers with some tape cells" },
  { MACHINEMETA, {"Mm", ""}, "",
     "show some meta information about the machine" },
  { BUFFERSTATE, {"bu", ""}, "",
     "show the state of the machine buffer (stack/workspace)" },
  { STACKSTATE, {"S", "stack"}, "",
     "show the state of the machine stack" },
  { TAPESTATE, {"T", "t.."}, "",
     "show the state of the machine tape" },
  { TAPEINFO, {"TT", "tapeinfo"}, "",
     "show detailed info of the state of the machine tape" },
  { TAPECONTEXTLESS, {"tcl", "lesstape"}, "",
     "reduce ammount of tape that will be displayed by printSomeTapeInfo()" },
  { TAPECONTEXTMORE, {"tcm", "moretape"}, "",
     "increase ammount of tape to be displayed by printSomeTapeInfo()" },
  { RESETINPUT, {"i.r", ""}, "",
     "reset the input stream" },
  { RESETMACHINE, {"M.r", ""}, "",
     "reset the machine to original state" },
  { STEPMODE, {"m.s", ""}, "",
     "make <enter> step through instructions" },
  { PROGRAMMODE, {"m.p", ""}, "",
     "make <enter> display program state" },
  { MACHINEMODE, {"m.m", ""}, "",
     "make <enter> display machine state" },
  { COMPILEMODE, {"m.c", ""}, "",
     "entered instructions are compiled but not executed" },
  { IPCOMPILEMODE, {"m.ipc", ""}, "",
     "entered instructions are compiled at current ip position" },
  { ENDCOMPILEMODE, {"m.ec", ""}, "",
     "entered instructions are compiled at end of program" },
  { INTERPRETMODE, {"m.int", "interpret"}, "",
     "entered instructions are executed but not compiled" },
  { LISTPROGRAM, {"ls", "p.ls"}, "",
     "list all instructions in the machines current program" },
  { LISTSOMEPROGRAM, {"l", "list"}, "",
     "list current instructions in the machines program" },
  { LISTPROGRAMWITHLABELS, {"pl", "p.ll"}, "",
     "list all instructions in program with labels (and jump labels)" },
  { PROGRAMMETA, {"pm", "pi"}, "",
     "show some meta information about the current program" },
  { SAVEPROGRAM, {"wa", "p.w"}, "<file>",
     "save the current program as 'assembler'" },
  { SHOWJUMPTABLE, {"jj", "showjumps"}, "",
     "Show the jumptable generated by buildJumpTable()" },
  { LOADPROGRAM, {"l.asm", "l.a"}, "<file>",
     "load machine assembler commands from text file" },
  { LOADASM, {"asm", "as"}, "",
     "load 'asm.pp' (the script parser) and reset the machine" },
  { LOADLAST, {"last", "p.ll"}, "",
     "load 'last.pp' (the program automatically saved on exit)" },
  { LOADSAVED, {"sav", "l.sav"}, "",
     "load 'sav.pp' (output of the 'write' command.)" },
  { LISTSAVFILE, {"lss", "ls.sav"}, "",
     "list the contents of 'sav.pp' (output of the 'write' command.)" },
  { SAVELAST, {"ww", "p.ww"}, "",
     "save 'last.pp' (the program automatically saved on exit)" },
  { CHECKPROGRAM, {"p.v", ""}, "",
     "validate or check the machines compiled program " },
  { CLEARPROGRAM, {"p.dd", "dd"}, "",
     "delete the machines compiled program " },
  { CLEARLAST, {"p.dl", "pdl"}, "",
     "delete the last instruction in the compiled program " },
  { INSERTINSTRUCTION, {"pi", "p.i"}, "",
     "insert an instruction at the current program ip " },
  { EXECUTEINSTRUCTION, {"n", "."}, "",
     "execute the next (current) compiled instruction in program" },
  { PARSEINSTRUCTION, {"pi:", "pi"}, "<test instruction text>",
     "parse some example text into a compiled instruction" },
  { TESTWRITEINSTRUCTION, {"twi", "twi"}, "",
     "shows how the current instruction will be written by writeInstruction()" },
  { STEPCODE, {"p.s", "ps"}, "",
     "step through the next instruction in compiled program" },
  { RUNCODE, {"rr", "p.r"}, "",
     "run the whole compiled program from the current instruction" },
  { RUNZERO, {"r0", "p.r0"}, "",
     "run the whole compiled program from instruction zero" },
  { RUNTOTRUE, {"rrt", "p.rt"}, "",
     "run the whole compiled program until flag is set to true" },
  { RUNTOWORK, {"rrw", "runwork"}, "<text>",
     "run program until the workspace is exactly the given text" },
  { RUNTOENDSWITH, {"rre", "runworkendswith"}, "<text>",
     "run program until the workspace ends with the given text" },
```

```c
  { IPZERO, {"p<<", "p0"}, "",
     "set the instruction pointer to zero" },
  { IPEND, {"p>>", "p.e"}, "",
     "set the instruction pointer to the end of the program" },
  { IPGO, {"pg", "p.g"}, "",
     "set the instruction pointer to the given instruction" },
  { IPPLUS, {"p>", "p.>"}, "",
     "increment the instruction pointer without executing" },
  { IPMINUS, {"p<", "p.<"}, "",
     "decrement the instruction pointer without executing" },
  { SHOWSTREAM, {"ss", "ss"}, "",
     "shows the next few characters from the input stream" },
  { EXIT, {"X", "exit"}, "",
     "exit the maching testing program" },
  { UNKNOWN, {"", ""}, "", ""}
};

// given the abbreviated command, this returns the command
// number (in the array and enumeration)
enum Command abbrevToCommand(char c) {
  int ii;
  for (ii = ADD; ii < UNDEFINED; ii++) {
    if (info[ii].abbreviation == c) return ii;
  }
  return UNDEFINED;
}

enum CommandType {JUMPS=0, TESTS, STACK, WORK, TAPE, ACCUMULATOR, OTHER};
// what type of command is it, a jump, test, stack, etc
enum CommandType commandType(enum Command com) {
  if (strncmp(info[com].name, "jump", 4) == 0)
    return JUMPS;
  if (strncmp(info[com].name, "test", 4) == 0)
    return TESTS;
  if (strcmp(info[com].name, "push") == 0)
    return STACK;
  if (strcmp(info[com].name, "pop") == 0)
    return STACK;
  if (strcmp(info[com].name, "get") == 0)
    return TAPE;
  if (strcmp(info[com].name, "put") == 0)
    return TAPE;
  if (strcmp(info[com].name, "a+") == 0)
    return ACCUMULATOR;
  if (strcmp(info[com].name, "a-") == 0)
    return ACCUMULATOR;
  if (strcmp(info[com].name, "zero") == 0)
    return ACCUMULATOR;
  if (strcmp(info[com].name, "cc") == 0)
    return ACCUMULATOR;
  if (strcmp(info[com].name, "ll") == 0)
    return ACCUMULATOR;
  if (strcmp(info[com].name, "until") == 0)
    return WORK;
  if (strcmp(info[com].name, "while") == 0)
    return WORK;
  if (strcmp(info[com].name, "whilenot") == 0)
    return WORK;
  if (strcmp(info[com].name, "read") == 0)
    return WORK;

  return OTHER;
}

// given a short or long machine command name, return the
// enum command value
enum Command textToCommand(const char * text) {
  int ii;
  for (ii = ADD; ii < UNDEFINED; ii++) {
    if ((strlen(text) == 1) && (info[ii].abbreviation == text[0]))
      return ii;
    else if (strcmp(text, info[ii].name) == 0)
      return ii;
  }
  return UNDEFINED;
}

// info functions:
void fprintCommandNames(FILE * file) {
  printf("Valid machine commands are [name (abbrev)]:\n ");
  int ii;
  for (ii = ADD; ii < UNDEFINED; ii++) {
    printf("%s ", info[ii].name);
  }
  printf("\n");
}

/* display help for one interactive help command
   (not a machine command. which may be should be referred to
   as instructions to avoid confusion) */
void printHelpCommand(int command, int comColour, int helpColour) {
  int ii = command;
  printf("%s%4s: %s%s%s ",
     colourInfo[comColour].ansi, testInfo[ii].names[0],
     colourInfo[helpColour].ansi,
     testInfo[ii].description, NORMAL);
}

/* Display help just for core help commands to assist the
   user to start to use the interactive system */
void printUsefulCommands() {
  int commands[] = {
    HELP, MACHINEPROGRAM, LISTMACHINECOMMANDS, EXECUTEINSTRUCTION,
    LISTPROGRAM, RUNCODE};

  printf("\nUseful interactive commands: \n"
         "---------------  \n");
  int nn;
  for (nn = 0; nn < 6; nn++) {
    printHelpCommand(commands[nn], YELLOWc, WHITEc); printf("\n");
  }
}

// display help for all commands, one per line
void machineCommandHelp() {
  printf("%s[All machine instructions]%s:\n", YELLOW, NORMAL);
  int ii;
  char key;
  for (ii = 0; ii < UNDEFINED; ii++) {
    printf(" %s%c%s - %s: %s%s%s \n",
      WHITE, info[ii].abbreviation, BLUE, info[ii].name, GREEN,
      info[ii].shortDesc, NORMAL);
    if ( (ii+1) % 14 == 0 ) {
      pagePrompt();
      key = getc(stdin);
      if (key == 'q') { return; }
    }
  }
  printf("\n");
}

// searches info array for machine commands matching a search term
void searchCommandHelp(char * text) {
  int ii;
  int jj = 0;
  printf("%s[Searching machine commands]%s:\n", YELLOW, NORMAL);
  for (ii = 0; ii < UNDEFINED; ii++) {
    if ((strstr(info[ii].shortDesc, text) != NULL) ||
        (strstr(info[ii].longDesc, text) != NULL) ||
        (strstr(info[ii].name, text) != NULL)) {
      jj++;

      printf(" %s%c%s - %s: %s \n",
        GREEN, info[ii].abbreviation, NORMAL, info[ii].name,
        info[ii].shortDesc );

      if ( (jj+1) % 14 == 0 ) {
        pagePrompt();
        getc(stdin);
      }
    } // if search term found
  } // for

  if (jj == 0) {
    printf("No results found for ’%s’\n", text);
  }
}

void showCommandNames() {
  printf("%s Valid machine commands are [name (abbrev)]: \n%s", BLUE, NORMAL);
  int ii;
  for (ii = ADD; ii < UNDEFINED; ii++) {
    printf(" %s %s(%s%c%s)%s",
      info[ii].name, GREY, YELLOW, info[ii].abbreviation, GREY, NORMAL);
    if ( (ii+1) % 4 == 0 ) { printf ("\n"); }
  }
  printf("\n");
}

// display each machine command and a one line description
// of what it does using bash colours
void fprintCommandSummary(FILE * file) {
  printf("A Summary of All Machine Commands:\n");
  printf("Format: name (abbreviation) title\n");
  int ii;
  for (ii = ADD; ii < UNDEFINED; ii++) {
    printf("%s\n %s", info[ii].name, info[ii].shortDesc);
  }
  printf("\n");
}

// display each machine command and a one line description
// of what it does using bash colours and paging a few lines at
// a time
void showCommandSummary() {
  printf("%sA Summary of All Machine Commands:\n%s ",BLUE,NORMAL);
  printf("%sFormat: name (abbreviation) title\n%s ",CYAN,NORMAL);
  int ii;
  for (ii = ADD; ii < UNDEFINED; ii++) {
    printf("%s%s (%c)%s\n  %s\n%s",
      BLUE, info[ii].name, info[ii].abbreviation,
      GREEN, info[ii].shortDesc, NORMAL);
    if ( (ii+1) % 10 == 0 ) {
      printf("any key to continue...");
      getc(stdin);
    }
  }
  printf("\n");
}

// print all information about a given command
void printCommandInfo(enum Command command) {
  printf(
    "command name: %s\n"
    "abbreviation: %c\n"
    "number of arguments: %d\n"
    "short description: %s\n"
    "long description: %s\n"
    "example: %s\n",
    info[command].name, info[command].abbreviation,
    info[command].args, info[command].shortDesc, info[command].longDesc,
    info[command].example
    );
}

// old
void printCommandNamesAndDescriptions() {
  printf("Valid commands are:\n ");
  int ii;
  for (ii = ADD; ii < NOP; ii++) {
    printf("%s;\n  %s\n", info[ii].name, info[ii].shortDesc);
  }
  printf("\n");
}


// the following enumeration and structure provide information
// about the standard c character classes, defined in ctype.h
// these classes will be used with the TESTCLASS command and
// also with the WHILE command. Parameters for Instructions will
// have a function pointer to the correct class function compiled
// into them, to speed up execution.

// each class corresponds to a "isUpper, isLower etc" function.

      //  fn = &isalnum;

enum Class {
  ALNUM=0, ALPHA, BLANK, CNTRL, DIGIT, GRAPH, LOWER, PRINTABLE,
  PUNCT, SPACE, UPPER, XDIGIT, NOCLASS };

// contain information about different character classes
struct {
  enum Class class;
  char * name;
  char * description;
  int (* classFn)(int);  // a function pointer for the ctype.h functions
} classInfo[] = {
  { ALNUM, "alnum", "alphanumeric like [0-9a-zA-Z]", isalnum},
  { ALPHA, "alpha", "alphabetic like [a-zA-Z]", isalpha},
  { BLANK, "blank", "blank chars, space and tab", isblank},
  { CNTRL, "cntrl", "control chars, ascii 000 to 037 and 177 (del)", iscntrl},
  { DIGIT, "digit", "digits 0-9", isdigit},
  { GRAPH, "graph", "graphical chars same as :alnum: and :punct:", isgraph},
  { LOWER, "lower", "lower case letters [a-z]", islower},
  { PRINTABLE, "print", "printable chars ie :graph: + space", isprint},
  { PUNCT, "punct", "punctuation ie !\"#$%&’()*+,-./:;<=>?@[\\]^_`{|}~.", ispunct},
  { SPACE, "space", "all whitespace, eg \\n\\r\\t vert tab, space, \\f", isspace},
  { UPPER, "upper", "upper case letters [A-Z]", isupper},
  { XDIGIT, "xdigit", "hexadecimal digit ie [0-9a-fA-F]", isxdigit,},
  { NOCLASS, "noclass", "", NULL},
};

// given a character class text return the enumerated constant
enum Class textToClass(const char * text) {
  int ii;
  for (ii = 0; ii < NOCLASS; ii++) {
    if (strncmp(text, classInfo[ii].name, strlen(classInfo[ii].name)) == 0)
      return ii;
  }
  return NOCLASS;
}

// given a pointer to a class function return the class
// a bit tricky ....
// a func pointer as argument to a function.
```

```c
//int add2to3(int (*functionPtr)(int, int)) {

enum Class fnToClass(void * func) {
  int ii;
  for (ii = 0; ii < NOCLASS; ii++) {
    if (func == classInfo[ii].classFn) return ii;
  }
  return NOCLASS;
}

// display all valid character classes
void showClasses() {
  printf("%sValid classes are: \n%s", CYAN, NORMAL);
  int ii;
  for (ii = 0; ii < NOCLASS; ii++) {
    if ( ii % 4 == 0 ) printf("\n  ");
    printf("%s%s%s - ", GREEN, classInfo[ii].name, GREY);
  }
  printf("\n");
  printf("%sClasses are used by the commands : \n%s", AQUA, NORMAL);
  printf("%s  while, whilenot and testclass \n%s", AQUA, NORMAL);
  printf("%seg: %s w [:space:] \n%s", BLUE, CYAN, NORMAL);
  printf("  reads the input stream while the peek register \n");
  printf("  is a whitespace character (space, tab, newline etc) \n");
}

// display help for all character classes, one per line
void printClasses() {
  printf("%sValid classes are: \n%s", CYAN, NORMAL);
  int ii;
  for (ii = 0; ii < NOCLASS; ii++) {
    printf(" %s%s%s - %s \n",
      GREEN, classInfo[ii].name, NORMAL, classInfo[ii].description);
    if ( (ii+1) % 14 == 0 ) {
      printf("\n%spress <enter> to continue...%s", CYAN, NORMAL);
      getc(stdin);
    }
  }
  printf("\n");
  printf("%sClasses are used by the commands : \n%s", AQUA, NORMAL);
  printf("%s  while, whilenot and testclass \n%s", AQUA, NORMAL);
  printf("%seg: %s w :space: \n%s", BLUE, CYAN, NORMAL);
  printf("  reads the input stream while the peek register \n");
  printf("  is a whitespace character (space, tab, newline etc) \n");

}

// parameters within a compiled instruction. They may have
// various data types; text numeric etc (eg: a jump to another instruction)
// hence the need for a union structure.
//

struct Parameter {
  //int datatype;
  //the datatype determines how the parameter will be used.
  //eg: int as a jump target, text as something to add to workspace,
  //    range in a test etc
  enum Type {
    TEXT,    // some string
    INT,     // integer number
    CLASS,   // a character class (eg alnum, alpha, digit, xdigit)
    RANGE,   // range of characters eg a-z
    LIST,    // list of characters eg abxy
    UNSET    // no value
  } datatype;

  union {
    char text[200];        // probably should be char * text, with malloc
    int number;            // eg for jumps (jump 6 - jump to instruction 6)
    int (* classFn)(int);  // a function pointer for the ctype.h functions
    char range[2];         // first, last
    char list[200];        // should be malloced
  };
};

/* this prints a compact display of the parameter suitable
   for including as part of printInstruction(). See showParameter()
   for a longer format */
int printParameter(struct Parameter parameter) {
  // if the parameter is not used, dont print it
  printf(" %s[%s", GREY, CYAN);

  switch (parameter.datatype) {
    case INT:
      printf("%sint:%s%d", BROWN, CYAN, parameter.number);
      break;
    case TEXT:
      printf("%stext:%s", BROWN, CYAN);
      escapeSpecial(parameter.list, CYAN);
      break;
    case CLASS:
      printf("%sclass:%s%s",
        BROWN, CYAN, classInfo[fnToClass(parameter.classFn)].name);
      break;
    case RANGE:
      printf("%srange:%s%c-%c", BROWN, CYAN,
        parameter.range[0], parameter.range[1]);
      break;
    case LIST:
      printf("%slist:%s", BROWN, CYAN);
      escapeSpecial(parameter.list, CYAN);
      break;
    default:
      printf("%s??error??:%s", BROWN, CYAN);
      break;
  }
  printf("%s]%s ", GREY, NORMAL);
  return parameter.datatype;
}

// one item in the label jump table (to convert asm labels to
// instruction/line numbers
struct Label {
  char text[64];       // label maximum 64 characters
  int instructNumber;  // line/instruction number equivalent for label
};

// a declaration
void printJumpTable(struct Label []);

// a compiled instruction parsed from the script text, or
// user input
// eg: add 'xx'
struct Instruction {
  enum Command command;
  struct Parameter a;
  struct Parameter b;
};

//struct Instruction * newInstruction(struct Instruction * ii, enum Command cc) {
void newInstruction(struct Instruction * ii, enum Command cc) {
  ii->command = cc;
  ii->a.datatype = UNSET;
  ii->b.datatype = UNSET;
}

// probably a memcpy of old and new would do the trick
// struct Instruction * copyInstruction(

void copyInstruction(
  struct Instruction * new, struct Instruction * old) {
  //memcpy(new, old, sizeof(new));
  new->command = old->command;
  memcpy(&new->a, &old->a, sizeof(struct Parameter));
  memcpy(&new->b, &old->b, sizeof(struct Parameter));
}


/*
display the instruction with parameters and parameter types.
*/
void debugInstruction(struct Instruction * ii) {
  printf("%s%9s %s[%s", AQUA, info[ii->command].name, GREY, CYAN);
  // also display "unset" datatype, with empty value
  /*
  switch (ii->a.datatype) {
    case INT:
      printf("%sint:%s%d", BROWN, CYAN, ii->a.number);
      break;
    case TEXT:
  }
  */
  if (ii->a.datatype == INT)
    printf("%sint:%s%d", BROWN, CYAN, ii->a.number);
  else if (ii->a.datatype == TEXT) {
    printf("%stext:%s", BROWN, CYAN);
    escapeSpecial(ii->a.list, CYAN);
  } else if (ii->a.datatype == CLASS)
    printf("%sclass:%s%s",
      BROWN, CYAN, classInfo[fnToClass(ii->a.classFn)].name);
  else if (ii->a.datatype == RANGE)
    printf("%srange:%s%c-%c", BROWN, CYAN, ii->a.range[0], ii->a.range[1]);
  else if (ii->a.datatype == LIST) {
    printf("%slist:%s", BROWN, CYAN);
    escapeSpecial(ii->a.list, CYAN);
  }
  else if (ii->a.datatype == UNSET)
    printf("%sunset%s", BROWN, CYAN);
  else
    printf("%s??error??:%s", BROWN, CYAN);

  printf("%s]%s ", GREY, NORMAL);

  // second parameter, but this is not used currently.
  printf("%s[%s", GREY, CYAN);
  if (ii->b.datatype == INT)
    printf("%d", ii->b.number);
  else if (ii->b.datatype == TEXT)
    printf("%s", ii->b.text);
  else if (ii->b.datatype == CLASS)
    printf("class:%s", classInfo[fnToClass(ii->b.classFn)].name);
  else if (ii->b.datatype == RANGE)
    printf("range:%c-%c", ii->a.range[0], ii->b.range[1]);
  else if (ii->b.datatype == LIST)
    printf("list:%s", ii->b.list);
  // no parameter so to nothing
  //else printf("");
  printf("%s]%s ", GREY, NORMAL);
}

/*
 show the instruction in a format suitable for program listings

 Use the escapeSpecial() function below to
 to display special chars (eg \n \r ...) as escaped sequences
 in a different colour to the rest of the text.

 todo:!!
  write function printParameter() to display a parameter
  with its datatype
*/
void printInstruction(struct Instruction * ii) {
  printf("%s%9s", AQUA, info[ii->command].name);
  if (ii->a.datatype != UNSET) {
    printParameter(ii->a);
  }
  printf(" ");
  if (ii->b.datatype != UNSET) {
    printParameter(ii->b);
  }
} // printInstruction

// convert special characters in a string to their escaped
// equivalent eg \n \r \v \f \t \? etc
// also escapes the given delimiter eg " or ]
//
// this has to be done char by char and is used in the
// saveAssembledProgram() function, writeInstruction() function etc
void escapeText(FILE * file, char * text) {
  char * cc = text;
  while (*cc != '\0') {
    switch (*cc) {
      case '"':
        fprintf(file, "\\\"");
        break;
      case ']':
        fprintf(file, "\\]");
        break;
      case '\\':
        fprintf(file, "\\\\");
        break;
      case '\n':
        fprintf(file, "\\n");
        break;
      case '\t':
        fprintf(file, "\\t");
        break;
      case '\r':
        fprintf(file, "\\r");
        break;
      case '\f':
        fprintf(file, "\\f");
        break;
      default:
        fprintf(file, "%c", *cc);
        break;
    } // switch cc
    cc++;
  } // while
} // fn

// this escapes special chars such as \n \r \\ and colours
// them differently and displays on stdout. This will be used
// by the printInstruction() function. This does not escape delimiter
// chars such as " and ]
void escapeSpecial(char * text, char * colour) {
  char * cc = text;
  // dont need this line, since normal characters
  // are handled below in the switch
  printf("%s", colour);
  while (*cc != '\0') {
    switch (*cc) {
      case '\n':
        printf("%s\\n%s", YELLOW, colour);
        break;
      case '\t':
        printf("%s\\t%s", YELLOW, colour);
        break;
      case '\r':
        printf("%s\\r%s", YELLOW, colour);
        break;
      case '\f':
        printf("%s\\f%s", YELLOW, colour);
        break;
      case '\v':
        printf("%s\\v%s", YELLOW, colour);
        break;
      default:
        printf("%s%c", colour, *cc);
        break;
    } // switch cc
    cc++;
  } // while
  printf("%s", NORMAL);
}

// show the instruction in a format suitable for program listings
// with special chars \n \r etc shown escaped in different colour
void printEscapedInstruction(struct Instruction * ii) {
  printf(" %s%s %s[%s", BLUE, info[ii->command].name, GREY, CYAN);
  if (ii->a.datatype == INT)
    printf("%d", ii->a.number);
  else if (ii->a.datatype == TEXT)
    escapeSpecial(ii->a.text, CYAN);
  else if (ii->a.datatype == CLASS)
    printf("%sclass:%s%s", BROWN, CYAN, classInfo[fnToClass(ii->a.classFn)].name);
  else if (ii->a.datatype == RANGE)
    printf("%srange:%s%c-%c", BROWN, CYAN, ii->a.range[0], ii->a.range[1]);
  else if (ii->a.datatype == LIST) {
    printf("%slist:", BROWN);
    escapeSpecial(ii->a.list, CYAN);
  }
  // else printf("");
  printf("%s]%s ", GREY, NORMAL);
  // deal with 2nd parameter ii->b ??
}

// this is used by saveAssembledProgram() or writeAssembledProgram()
// special and delim chars are escaped, eg: \n \r \] \"
//
void writeInstruction(struct Instruction * ii, FILE * file) {
  fprintf(file, "%s ", info[ii->command].name);
  switch (ii->a.datatype) {
    case INT:
      fprintf(file, "%d", ii->a.number);
      break;
    case TEXT:
      // special chars \n \r \t \\ and " need to be escaped
      fprintf(file, "\"");
      escapeText(file, ii->a.text);
      fprintf(file, "\"");
      break;
    case CLASS:
      fprintf(file, "[:%s:]", classInfo[fnToClass(ii->a.classFn)].name);
      break;
    case RANGE:
      // todo: check range arguments (escaped characters???)
      // no I dont think ranges should have escaped characters
      fprintf(file, "[%c-%c]", ii->a.range[0], ii->a.range[1]);
      break;
    case LIST:
      // special chars \n \r \t \\ etc and ] need to be escaped
      fprintf(file, "[");
      escapeText(file, ii->a.list);
      fprintf(file, "]");
      break;
    default:
      // no parameter
      // fprintf(file, "");
      break;
  }
  // deal with 2nd parameter, but this is not currently used ....

}

// print to file the instruction with a line number and colours
// this function is used in the validateProgram function to display
// or log program errors. We also need something like this in
// saveAssembledProgram()
//
void numberedInstruction(
  struct Instruction * instruction, int lineNumber, FILE * file) {
  fprintf(file, "%d: %s ", lineNumber, info[instruction->command].name);
  if (instruction->a.datatype == INT)
    fprintf(file, "[%d] ", instruction->a.number);
  else if (instruction->a.datatype == TEXT)
    fprintf(file, "[%s] ", instruction->a.text);
  else printf("[] ");

  if (instruction->b.datatype == INT)
    fprintf(file, "[%d] ", instruction->b.number);
  else if (instruction->b.datatype == TEXT)
    fprintf(file, "[%s] ", instruction->b.text);
  else printf("[] ");
  printf("\n");
}



// the compiled program, which is a set of instructions
// the instruction listing really needs to be malloced dynamically
// not a static array
#define PROGRAMCAPACITY 500
struct Program {
  // date and time when compiled
  time_t compileDate;
  // how long program took to compile (milliseconds), timed with clock() ?
  clock_t compileTime;
  // starting to execute
  int startExecute;
  // time ended executing
  int endExecute;
  // whether new compiled instructions are appended to the
  // program (after count), inserted after IP or overwrite from ip
  // ??? necessary.
  enum CompileMode {APPEND, INSERT, OVERWRITE} compileMode;
  // how much room for instructions
  size_t capacity;
  // how many times has program memory been reallocated
  int resizings;
  //how many instructions are in the program
  int count;
  //current instruction (next to be executed)
  int ip;
  // the set of program instructions (dynamically allocated)
  struct Instruction * listing;
  // if applicable, name of assembly file containing instructions
  char source[128];

  // an array of line labels and line numbers from assembly listing
  struct Label labelTable[256];

  // static allocation of memory for instructions
  // struct Instruction listing[PROGRAMCAPACITY];
};

/* a prototype declaration, just to stop a gcc compiler
   warning, because I am using compile() before I define it.
*/
int compile(struct Program *, char *, int, struct Label[]);

void newProgram(struct Program * program, size_t capacity) {
  program->resizings = 0;
  program->count = 0;
  program->ip = 0;
  program->compileTime = -1;
  program->compileDate = -1;
  program->startExecute = -1;
  program->endExecute = -1;
  program->listing = malloc(capacity * sizeof(struct Instruction));
  if(program->listing == NULL) {
    fprintf(stderr,
      "couldnt allocate memory for program listing newProgram()\n");
    exit(EXIT_FAILURE);
  }
  program->capacity = capacity;
  int ii;
  for (ii = 0; ii < capacity; ii++) {
    newInstruction(&program->listing[ii], UNDEFINED);
  }
  strcpy(program->source, "?");
  memset(program->labelTable, 0, sizeof(program->labelTable));
}

void freeProgram(struct Program * pp) {
  int ii;
  for (ii = 0; ii < pp->capacity; ii++) {
    // If instruction parameters are malloced, as they
    // should be, then we will have to free the associated
    // memory. But at the moment it is static memory allocation.
    //freeInstruction(&program->listing[ii]);
  }
  free(pp->listing);
}

// increase program capacity
// there is a bug in the capacity arithmetic so that
// on the second realloc 2 instructions are wiped.
void growProgram(struct Program * program, size_t increase) {
  printf("Program listing is growing!!\n");
  program->capacity = program->capacity + increase;
  program->listing = realloc(program->listing,
    program->capacity * sizeof(struct Instruction));
  if(program->listing == NULL) {
    fprintf(stderr,
      "couldnt allocate more memory for program listing in growProgram()\n");
    exit(EXIT_FAILURE);
  }
  int ii;
  for (ii = program->count; ii < program->capacity; ii++) {
    newInstruction(&program->listing[ii], UNDEFINED);
  }
}

// insert an instruction at the current ip position
void insertInstruction(struct Program * program) {
  int ii;
  struct Instruction * thisInstruction;
  if (program->count == program->capacity) {
    printf("no more room in program... \n");
    return;
  }
  for (ii = program->count-1; ii >= program->ip; ii--) {
    thisInstruction = &program->listing[ii];
    if (commandType(thisInstruction->command) == JUMPS) {
       // bug!! actually we only increment jumps if
       // the jump target is after the instruction
       if (thisInstruction->a.number > program->ip) {
         thisInstruction->a.number++;
       }
    }
    //printInstruction(thisInstruction); printf(" \n");
    copyInstruction(&program->listing[ii+1], thisInstruction);
  }
  newInstruction(&program->listing[program->ip], NOP);
  program->count++;
}

/* print meta information about the program
   This information is part of the Program structure, so
   it is not really meta information, technically.
*/
void printProgramMeta(struct Program * program) {
  char Colour[30];
  char date[30];
  char time[30];
  strcpy(date, "?");
  strcpy(time, "?");
  strcpy(Colour, BROWN);
  printf("%s         Program source:%s %s \n",
    Colour, NORMAL, program->source);
  printf("%sCapacity (Instructions):%s %ld \n",
    Colour, NORMAL, program->capacity);
  printf("%s    Memory reallocation:%s %d \n",
    Colour, NORMAL, program->resizings);
  printf("%s  How many instructions:%s %d \n",
    Colour, NORMAL, program->count);
  printf("%s    Current instruction:%s %d  instruction:",
    Colour, NORMAL, program->ip);
  printInstruction(&program->listing[program->ip]);
  printf("\n");
  if (program->compileDate != -1) {
    strcpy(date, ctime(&program->compileDate));
  }
  if (program->compileTime != -1) {
    sprintf(time, "%ld", program->compileTime);
  }
  printf("%s            Compiled at:%s %s \n", Colour, NORMAL, date);
  printf("%s           Compile time:%s %s milliseconds \n",
    Colour, NORMAL, time);
}

/* given a text label get the line/instruction number from the table
   or else return -1
*/
int getJump(char * label, struct Label table[]) {
  int ii;
  for (ii = 0; ii < 1000; ii++) {
     // if not found
     if (table[ii].text[0] == 0) return -1;
     // return number if label found
     if (strcmp(table[ii].text, label) == 0) {
       return table[ii].instructNumber;
     }
  }
  return -1;
}

/*
 return label for given instruction, or else null
 an empty string means "not found"
```

```c
*/
char * getLabel(int instruction, struct Label table[]) {
  int ii = 0;
  for (ii = 0; ii < 1000; ii++) {
     // if not found
     if (table[ii].text[0] == 0) break;
     if (table[ii].instructNumber == instruction) {
       return table[ii].text;
     }
  }
  return table[ii].text;
}

// print the instructions in a program
void printProgram(struct Program * program) {
  int ii;
  char key;
  struct Instruction * thisInstruction;
  printf("%sFull Program Listing %s", CYAN, NORMAL);
  printf(" %s(%ssize:%s%d%s ip:%s%d%s cap:%s%lu%s)%s \n",
    GREY, GREEN, NORMAL, program->count, GREEN, NORMAL,
    program->ip, GREEN, NORMAL, program->capacity, GREY, NORMAL);
  for (ii = 0; ii < program->count; ii++) {
    // page the listing
    if ( (ii+1) % 16 == 0 ) {
      printf("\n%s<enter>%s for more (%sq%s = exit):",
        AQUA, NORMAL, AQUA, NORMAL);
      key = getc(stdin);
      if (key == 'q') { return; }
      // go back a page? lets not worry about it
      // if (key == 'b') { ii = (((ii-16)>(0))?(ii-16):(0)); }
    }
    thisInstruction = &program->listing[ii];
    if (ii == program->ip) {
      printf("%s%3d> ", YELLOW, ii);
      printInstruction(thisInstruction);
      printf("%s\n", NORMAL);
    }
    else {
      printf("%s%3d:%s ", WHITE, ii, NORMAL);
      printInstruction(thisInstruction);
      printf("%s\n", NORMAL);
    }
  }
}


/* print all the instructions in a compiled program along
   with the labels which were parse from the assembly listing
   This may be handy for debugging jump target problems.
   Some jump targets may be relative, no?? */
void printProgramWithLabels(struct Program * program) {
  int ii;
  char key;
  char * label;
  struct Instruction * thisInstruction;
  printf("%s[Full Program Listing] %s", YELLOW, NORMAL);

  printf(" %s(%ssize:%s%d%s ip:%s%d%s cap:%s%lu%s)%s \n",
    GREY, GREEN, NORMAL, program->count, GREEN, NORMAL,
    program->ip, GREEN, NORMAL, program->capacity, GREY, NORMAL);

  /* to do better paging, we can change this to a while loop
     and increment ii or set ii=0 (g) ii=max (G) etc */
  for (ii = 0; ii < program->count; ii++) {
    // page the listing
    if ( (ii+1) % 16 == 0 ) {
      printf("\n%s<enter>%s for more (%sq%s = exit) ...",
        AQUA, NORMAL, AQUA, NORMAL);
      key = getc(stdin);
      // make 'g' go to top, 'G' go to bottom etc
      if (key == 'q') { return; }
      // go back a page here, but how to do it?
      // if (key == 'b') { ii = (((ii-16)>(0))?(ii-16):(0)); }
    }
    thisInstruction = &program->listing[ii];
    // check if there is a label

    label = getLabel(ii, program->labelTable);
    if (strlen(label) > 0) {
      printf("%s:\n", label);
    }
    if (ii == program->ip) {
      printf("%s%3d> ", YELLOW, ii);
      printInstruction(thisInstruction);
      printf("%s\n", NORMAL);
    }
    else {
      printf("%s%3d:%s ", WHITE, ii, NORMAL);
      printInstruction(thisInstruction);
      printf("%s\n", NORMAL);
    }
  }
}

// show the program around the ip instruction pointer
void printSomeProgram(struct Program * program, int context) {
  int ii;
  struct Instruction * thisInstruction;
  int start = program->ip - context;
  int end = program->ip + context;
  if (start < 0) start = 0;
  if (end > program->capacity) end = program->capacity;
  //if (program->ip > end) end = program->ip + 2;
  printf("%sPartial Program Listing%s", CYAN, NORMAL);
  printf(" %s(%ssize:%s%d%s ip:%s%d%s cap:%s%lu%s)%s \n",
    GREY, GREEN, NORMAL, program->count, GREEN, NORMAL,
    program->ip, GREEN, NORMAL, program->capacity, GREY, NORMAL);
  for (ii = start; ii < end; ii++) {
    thisInstruction = &program->listing[ii];
    if (ii == program->ip) {
      printf(" %s%d> ", YELLOW, ii);
      printInstruction(thisInstruction);
      printf("%s\n", NORMAL);
    }
    else {
      printf(" %s%d%s: ", CYAN, ii, NORMAL);
      printInstruction(thisInstruction);
      printf("%s\n", NORMAL);
    }
  }
}

// save the compiled program as assembler to a text file
// we need to escape quotes in arguments !!
// also need to handle other parameter datatypes such as class,
// range, list etc.

// Line numbers may make modifying the assembly laborious.
void saveAssembledProgram(struct Program * program, FILE * file) {
  int ii;
  fprintf(file, "# Assembly listing \n");
  for (ii = 0; ii < program->count; ii++) {
    //fprintf(file, "%d: ", ii);
    writeInstruction(&program->listing[ii], file);
    fprintf(file, "\n");
  }
  fprintf(file, "# End of program. \n");
  printf("%s%d%s instructions written to file. \n", BLUE, ii, NORMAL);
}

// in some cases it may be useful to have a numbered code listing
void saveNumberedProgram(struct Program * program, FILE * file) {
  int ii;
  fprintf(file, "# Numbered Assembly listing \n");
  for (ii = 0; ii < program->count; ii++) {
    fprintf(file, "%d: ", ii);
    writeInstruction(&program->listing[ii], file);
    fprintf(file, "\n");
  }
  fprintf(file, "# End of program. \n");
  printf("%s%d%s instructions written to file. \n", BLUE, ii, NORMAL);
}

// clear the machines compiled program
void clearProgram(struct Program * program) {
  int ii;
  for (ii = 0; ii < program->count + 50; ii++) {
    program->listing[ii].command = UNDEFINED;
    program->listing[ii].a.number = 0;
    program->listing[ii].a.datatype = UNSET;
    program->listing[ii].b.number = 0;
    program->listing[ii].b.datatype = UNSET;
  }
  program->count = 0;
  program->ip = 0;
  program->compileTime = -1;
  program->compileDate = 0;
}

/*
  scans an instruction parameter and stops at the delimiter
  also converts escape sequences such as \n \r to their
  actual character. The unescaped sequence is stored starting
  at pointer param.
  return the new position in the scan stream. The datatype of
  the parameter is set by the instructionFromText() function
*/
char * scanParameter(char * param, char * arg, int line) {
  char * nextChar;
  char delimiter;

  if (strlen(arg) == 0) return arg;

  // lets skip leading spaces here. Although this skips
  // all leading characters not just spaces.
  char * found;
  found = arg + strcspn(arg, "[({\"'");
  // no delimiter found, so bail
  if (found == '\0') return arg;

  arg = found;
  if (*arg == '[') delimiter = ']';
  else if (*arg == '(') delimiter = ')';
  else if (*arg == '{') delimiter = '}';
  else delimiter = *arg;   // eg: " or ' etc

  nextChar = arg+1;
  while ((*nextChar != '\0') && (*nextChar != delimiter)) {
    if (*nextChar == '\\') {
      nextChar++;
        // check \n \t \] \\ etc
      switch (*nextChar) {
        case 'n':
          *param = '\n';
          break;
        case 't':
          *param = '\t';
          break;
        case 'r':
          *param = '\r';
          break;
        case 'f':
          *param = '\f';
          break;
        case 'v':
          *param = '\v';
          break;
        default:
          // handle all other \\ slash escapes eg \\ \] \"
          *param = *nextChar;
          break;
      } // switch
    }
    else {
      *param = *nextChar;
    }
    *(param+1) = 0;
    nextChar++;
    param++;
  } // while not delimiter

  if (*nextChar != delimiter) {
    printf("%s Warning: %s on line %s%d%s. \n",
      PURPLE, NORMAL, YELLOW, line, NORMAL);
    printf("%s >> %s%s %s\n", BLUE, GREEN, arg, NORMAL);
    printf("  No terminating %s%c%s character for parameter \n",
      YELLOW, delimiter, NORMAL);
  }
  return nextChar;
} // scanParameter

/* print out the label table, just to make sure that it is
   getting built correctly. this table can be, and is, a property
   of the Program structure */
void printJumpTable(struct Label table[]) {
  int ii;
  printf("%s[Program label table]%s:\n", YELLOW, NORMAL);
  for (ii = 0; ii < 1000; ii++) {
     if (table[ii].text[0] == 0) break;
     printf("%s%15s:%s %d \n",
       BROWN, table[ii].text, NORMAL, table[ii].instructNumber);
  }
}

/* extracts the correct values for a parameter from a text argument
   it returns a pointer to the last character scanned in "args" or
   else NULL if there is no parameter. */
char * parameterFromText(
  FILE * file, struct Instruction * instruction, struct Parameter * param,
  char * args, int lineNumber, struct Label table[]) {
  char command[1001] = "";
  //char args[1001] = "";
  char label[64] = "";       // possible jump label
  char charclass[20] = "";   // eg space, alnum, print
  int position;     // calculate how many chars were scanned

  // here check if instruction is a jump (ie jumptrue/false/jump)
  // now check if it already has an integer argument. If not
  // check if 1st argument is a label. If so, convert to line number
  // using jumpTable[] and then convert to offset (line number - current
  // line/instruction number)

  if (commandType(instruction->command) == JUMPS) {
    // format "jump 32"
    if (1 == sscanf(args, "%d%n", &param->number, &position)) {
      param->datatype = INT;
      return args + position;
    }
    // format "jump label"
    else if (1 == sscanf(args, "%s%n", label, &position)) {
      // for debugging see the label jump table
      // printJumpTable(table);
      // find the label,
      int jump = -1;
      jump = getJump(label, table);
      // printf("<label: %s, jump: %d> \n", label, jump);
      if (jump == -1) {
        fprintf(file, "%s Parse Error: %s on line %s%d%s. \n",
          PURPLE, NORMAL, YELLOW, lineNumber, NORMAL);
        fprintf(file, "%s >> %s%s %s\n", BLUE, GREEN, args, NORMAL);
        fprintf(file, " Label '%s%s%s' not found. \n",
          YELLOW, label, NORMAL);
        fprintf(file,
          " Labels should start line and end with a \n"
          " colon (:). They are case sensitive. Jump targets should\n"
          " not include the colon. Labels cannot start with a number \n"
          " (because it is parsed as a line number).\n"
          " Type '%s%s%s' in interactive mode to see the label table\n",
             GREEN, testInfo[SHOWJUMPTABLE].names[0], NORMAL);

        return NULL;
      } else {
        //printf("command: %s \n", info[instruction->command].name);
        //printf("jump target=%d, linenumber=%d\n", jump, lineNumber);

        param->datatype = INT;
        if (instruction->command == JUMP) {
          param->number = jump;
        } else {
          param->number = jump - lineNumber;
        }
      }
      return args + position;
    }
  }

  // format "jump 32"
  // we need to use this position variable and return it
  // this is because we will call parameterFromText() again, to see
  // there are any more parameters.
  //int pos
  //(1 == sscanf(expression, "%lf%n", &value, &pos))
  if (1 == sscanf(args, "%d%n", &param->number, &position)) {
    param->datatype = INT;
    return args + position;
  }

  // format "while [:space:]
  else if ((args[0] == '[') && (args[1] == ':')) {
    if (sscanf(args+2, "%20[^:]", charclass) == 0) {

      fprintf(file, "%s Error: %s on line %s%d%s. \n",
        PURPLE, NORMAL, YELLOW, lineNumber, NORMAL);
      fprintf(file, "%s >> %s%s %s\n", BLUE, GREEN, args, NORMAL);
      fprintf(file, "  In argument %s%s%s, no character class given \n",
        YELLOW, args, NORMAL);
      return NULL;

    }
    else if (textToClass(charclass) == NOCLASS) {

      fprintf(file, "%s Error: %s on line %s%d%s of source file. \n",
        PURPLE, NORMAL, YELLOW, lineNumber, NORMAL);
      fprintf(file, "%s >> %s%s %s\n", BLUE, GREEN, args, NORMAL);
      fprintf(file, "  The character class %s%s%s is not valid \n",
        YELLOW, charclass, NORMAL);
      return NULL;

    }
    param->datatype = CLASS;
    param->classFn = classInfo[textToClass(charclass)].classFn;
    return 0;
  }

  // format "whilenot  [a-z]
  else if ((args[0] == '[') && (args[2] == '-')) {
    param->datatype = RANGE;
    param->range[0] = args[1];
    param->range[1] = args[3];
    return 0;
  }
  else if (args[0] == '[') {
    // bracket delimited parameter
    param->datatype = LIST;
    scanParameter(param->list, args, lineNumber);
  }
  else if (args[0] == '"') {
    // quote delimited parameter
    param->datatype = TEXT;
    char * next = scanParameter(param->text, args, lineNumber);
  }
  else if (args[0] == '{') {
    // brace delimited parameter
    param->datatype = TEXT;
    scanParameter(param->text, args, lineNumber);
  } // what sort of argument

  // print warnings if, for example, jump does not have an integer target
  // checkInstruction(instruction, stdout);
  return 0;
}
/*
   fills out an instruction given valid text. Also handles unescaping
   of special characters and delimiter characters
     eg:  add " this \r \t \" \: \\ "
     eg:  while [ghi \];;\\ ]

   the line/instruction number parameter is for error messages
   returns -1 when an error occurs
   add struct label table[] jumptable parameter.
*/
int instructionFromText(
  FILE * file, struct Instruction * instruction,
  char * text, int lineNumber, struct Label table[]) {
  char command[1001] = "";
  char args[1001] = "";
  char label[64] = "";       // possible jump label
  char charclass[20] = "";   // eg space, alnum, print

  sscanf(text, "%1000s %2000[^\n]", command, args);
  //printf("parsed - command=%s, args=%s \n", command, args);

  if (command[0] == '\0') {
    fprintf(file, "%s Parse Error: %s on line %s%d%s \n",
      PURPLE, NORMAL, YELLOW, lineNumber, NORMAL);
    fprintf(file, "%s >> %s%s %s\n", BLUE, GREEN, text, NORMAL);
    fprintf(file, " Instruction command missing \n");
    return -1;
  }

  if (textToCommand(command) == UNDEFINED) {
    fprintf(file, "%s Parse Error: %s on line %s%d%s. \n",
      PURPLE, NORMAL, YELLOW, lineNumber, NORMAL);
    fprintf(file, "%s >> %s%s %s\n", BLUE, GREEN, text, NORMAL);
    fprintf(file, " Command %s%s%s is not a valid instruction command \n",
      YELLOW, command, NORMAL);
    return -1;
  }

  instruction->command = textToCommand(command);

  if ((info[instruction->command].args > 0) && (args[0] == '\0')) {
    fprintf(file, "%s Error: %s on line %s%d%s of source file. \n",
      PURPLE, NORMAL, YELLOW, lineNumber, NORMAL);
    fprintf(file, "%s >> %s%s %s\n", BLUE, GREEN, text, NORMAL);
    fprintf(file, "  Command %s%s%s requires %s%d%s argument(s) but none \n"
           "  is given in the assembly file \n",
      YELLOW, command, NORMAL,
      YELLOW, info[instruction->command].args, NORMAL);
  }

  parameterFromText(
    file, instruction, &instruction->a, args, lineNumber, table);

  //FILE * file, struct Instruction * instruction,
  //char * text, int lineNumber, struct Label table[]) {

  // try to get the next parameter (for replace)
  // here check if instruction is a jump (ie jumptrue/false/jump)
  // now check if it already has an integer argument. If not
  // check if 1st argument is a label. If so, convert to line number
  // using jumpTable[] and then convert to offset (line number - current
  // line/instruction number)

  /*

  if (commandType(instruction->command) == JUMPS) {
    // format "jump 32"
    if (sscanf(args, "%d", &instruction->a.number) > 0) {
      instruction->a.datatype = INT;
      return 0;
    }
    // format "jump label"
    else if (sscanf(args, "%s", label) > 0) {
      // for debugging see the label jump table
      // printJumpTable(table);
      // find the label,
      int jump = -1;
      jump = getJump(label, table);
      // printf("<label: %s, jump: %d> \n", label, jump);
      if (jump == -1) {
        fprintf(file, "%s Parse Error: %s on line %s%d%s. \n",
          PURPLE, NORMAL, YELLOW, lineNumber, NORMAL);
        fprintf(file, "%s >> %s%s %s\n", BLUE, GREEN, text, NORMAL);
        fprintf(file, " Label '%s%s%s' not found. \n",
          YELLOW, label, NORMAL);
        fprintf(file,
          " Labels should start line and end with a \n"
          " colon (:). They are case sensitive. Jump targets should\n"
          " not include the colon. Labels cannot start with a number \n"
          " (because it is parsed as a line number).\n"
          " Type '%s%s%s' in interactive mode to see the label table\n",
             GREEN, testInfo[SHOWJUMPTABLE].names[0], NORMAL);

        return -1;
      } else {
        //printf("command: %s \n", info[instruction->command].name);
        //printf("jump target=%d, linenumber=%d\n", jump, lineNumber);

        instruction->a.datatype = INT;
        if (instruction->command == JUMP) {
          instruction->a.number = jump;
        } else {
          instruction->a.number = jump - lineNumber;
        }
      }
      return 0;
    }
  }

  // format "jump 32"
  if (sscanf(args, "%d", &instruction->a.number) > 0) {
    instruction->a.datatype = INT;
    return 0;
  }

  // format "while [:space:]
  else if ((args[0] == '[') && (args[1] == ':')) {
    if (sscanf(args+2, "%20[^:]", charclass) == 0) {
      fprintf(file, "%s Error: %s on line %s%d%s. \n",
        PURPLE, NORMAL, YELLOW, lineNumber, NORMAL);
      fprintf(file, "%s >> %s%s %s\n", BLUE, GREEN, text, NORMAL);
      fprintf(file, "  In argument %s%s%s, no character class given \n",
        YELLOW, args, NORMAL);
      return 1;
    }
    else if (textToClass(charclass) == NOCLASS) {
      fprintf(file, "%s Error: %s on line %s%d%s of source file. \n",
        PURPLE, NORMAL, YELLOW, lineNumber, NORMAL);
      fprintf(file, "%s >> %s%s %s\n", BLUE, GREEN, text, NORMAL);
      fprintf(file, "  The character class %s%s%s is not valid \n",
        YELLOW, charclass, NORMAL);
      return 1;
    }
    instruction->a.datatype = CLASS;
    instruction->a.classFn = classInfo[textToClass(charclass)].classFn;
    return 0;
  }

  // format "whilenot  [a-z]
  else if ((args[0] == '[') && (args[2] == '-')) {
    instruction->a.datatype = RANGE;
    instruction->a.range[0] = args[1];
    instruction->a.range[1] = args[3];
    return 0;
  }
  else if (args[0] == '[') {
    // bracket delimited parameter
    instruction->a.datatype = LIST;
    scanParameter(instruction->a.list, args, lineNumber);
  }
  else if (args[0] == '"') {
    // quote delimited parameter
    instruction->a.datatype = TEXT;
    char * next = scanParameter(instruction->a.text, args, lineNumber);
    // try to get the next parameter (for replace)
    if (strlen(next) != 0) {
      //printf("args=%s \nnext=%s\n", args,next);
      //scanParameter(instruction->b.text, next, lineNumber);
      //printf("args=%s \nnext=%s\n", args,next);
      //next = scanParameter(instruction->b.text, next, lineNumber);
    }
  }
  else if (args[0] == '{') {
    // brace delimited parameter
    instruction->a.datatype = TEXT;
    scanParameter(instruction->a.text, args, lineNumber);
  } // what sort of argument

  // print warnings if, for example, jump does not have an integer target
  // checkInstruction(instruction, stdout);
```

```c
  */
  return 0;
}

/* just build an array with label line numbers
   returns the number of labels found
*/
int buildJumpTable(struct Label table[], FILE * file) {
  char buffer[1000];
  char * line;
  int lineNumber;
  char text[2001];
  char label[64]; // hold the asm label
  int ii = 0;     // instruction number
  int ll = 0;     // label number

  while (fgets(buffer, 999, file) != NULL) {
    // printf("%s", line);
    line = buffer;
    line[strlen(line) - 1] = '\0';
    text[0] = '\0';
    lineNumber = -1;

    // Trim leading space
    while (isspace((unsigned char)*line)) line++;
    // skip blank lines
    if (*line == 0) continue;
    // lines starting with # are comments
    // if (line[0] == '#') continue;
    // lines starting with # are 1 line comments
    // lines starting with #* are multiline comments (to *#)
    if (line[0] == '#') {
      if (strlen(line) == 1) continue;
      if (line[1] == '*') {

        line = line + 2;
        //printf("multiline comment! %s", line);
        // multiline comment #* ... need to search for next *#
        if(strstr(line, "*#") != NULL) {
          line = strstr(line, "*#") + 2;
        } else {
          // search next lines for *#
          while (fgets(buffer, 999, file) != NULL) {
            line = buffer;
            line[strlen(line) - 1] = '\0';
            if (strstr(line, "*#") != NULL) {
              line = strstr(line, "*#") + 2;
              break;
            }
          }
        }
      } else continue;
    }

    sscanf(line, "%d: %2000[^\n]", &lineNumber, text);
    //debug: check that the arguments are getting parsed properly
    //printf("parsed - lineNumber=%d, text=%s \n", lineNumber, text);

    if (lineNumber == -1) {
      // if no line number, parse anyway
      sscanf(line, "%s", text);
      // only whitespace on line, so skip it
      if (text[0] == '\0') { continue; }
      sscanf(line, "%2000[^\n]", text);
    }

    // skip empty lines
    if (text[0] == '\0') { continue; }
```

```c
    // handle assembler labels using the jumpTable
    // labels are lines ending in ':'
    if (text[strlen(text) - 1] == ':') {
      sscanf(text, "%64[^:]:", label);
      // printf("New Label '%s' at instruction %d \n", label, ii);
      table[ll].instructNumber = ii;
      strcpy(table[ll].text, label);
      ll++;     // increment label number
      continue;
    }
    // increment instruction number
    ii++;
  } // while
  return ll;
}

/*
 load a program from an assembler listing in a text file
 uses compile() >> instructionFromText() or parseInstruction()
 maybe change this to ...(struct Machine * mm, ...) ??.
 The compiled instructions are inserted in the program at position
 "start".
*/
void loadAssembledProgram(struct Program * program, FILE * file, int start) {
  clearProgram(program);
  // this is not robust, we need to deal with very long
  // instructions using malloc (eg "add <lots of text>")
  char buffer[1000];
  char * line;
  int lineNumber;
  char text[2001];
  clock_t startTime = 0;
  clock_t endTime = 0;

  //int ii = 0;
  int ii = start;
  // build a table of label line numbers
  // printf("Put %s%d%s labels into the jump table. \n", YELLOW, ll, NORMAL);
  // this returns the number of labels found
  buildJumpTable(program->labelTable, file);

  // printJumpTable(program->labelTable);

  rewind(file);

  //struct Instruction * instruction;

  while (fgets(buffer, 999, file) != NULL) {
    // printf("%s", line);
    line = buffer;
    line[strlen(line) - 1] = '\0';
    text[0] = '\0';
    lineNumber = -1;
    // Trim leading space
    while (isspace((unsigned char)*line)) line++;
    // skip blank lines
    if (*line == 0) continue;

    // lines starting with # are 1 line comments
    // lines starting with #* are multiline comments (to *#)
    if (line[0] == '#') {
      if (strlen(line) == 1) continue;
      if (line[1] == '*') {

        line = line + 2;
        //printf("multiline comment! %s", line);
        // multiline comment #* ... need to search for next *#
        if(strstr(line, "*#") != NULL) {
          line = strstr(line, "*#") + 2;
        } else {
          // search next lines for *#
          while (fgets(buffer, 999, file) != NULL) {
            line = buffer;
            line[strlen(line) - 1] = '\0';
            if (strstr(line, "*#") != NULL) {
              line = strstr(line, "*#") + 2;
              break;
            }
          }
        }
      } else continue;
    }

    // skip label lines. improve this to trim spaces
    if (line[strlen(line)-1] == ':') continue;

    sscanf(line, "%d: %2000[^\n]", &lineNumber, text);
    //debug: check that the arguments are getting parsed properly
    //printf("parsed - lineNumber=%d, text=%s \n", lineNumber, text);

    if (lineNumber == -1) {
      // if no line number, parse anyway
      sscanf(line, "%s", text);
      // only whitespace on line, so skip it
      if (text[0] == '\0') { continue; }
      sscanf(line, "%2000[^\n]", text);
    }

    // empty line or just line number, just skip it
    if (*text == 0) continue;
    compile(program, text, ii, program->labelTable);
    ii++;
  } // while
  program->ip = 0;
  endTime = clock();
  program->compileTime = (double)((endTime - startTime)*1000) / CLOCKS_PER_SEC;
  program->compileDate = time(NULL);

  /*
  printf("Compiled %s%d%s instructions from '%s%s%s' in about %s%ld%s milliseconds
\n",
    CYAN, program->count, NORMAL,
    CYAN, program->source, NORMAL,
    CYAN, program->compileTime, NORMAL);
  */

}

// check datatype of instruction etc
int checkInstruction(struct Instruction * ii, FILE * file) {

  if ((info[ii->command].args > 0) && (ii->a.datatype == UNSET))  {
    fprintf(file, "error: missing argument for command \n");
  }
  if ((commandType(ii->command) == JUMPS) && (ii->a.datatype != INT))  {
    //printf(file, "error: non integer \n");
  }
  if ((info[ii->command].args == 0) && (ii->a.datatype != UNSET))  {
    fprintf(file, "warning: superfluous argument for command %s%s%s \n",
        YELLOW, info[ii->command].name, NORMAL);
  }

  switch (ii->command) {
    case ADD:
      if (ii->a.datatype != TEXT) {
        fprintf(file, "Error: ADD requires 'text' datatype \n");
      }
      if (*ii->a.text == 0) {
        fprintf(file, "No text parameter for ADD \n");
      }
      break;
    case CLIP:
      break;
    case CLOP:
      break;
    case CLEAR:
      break;
    case PRINT:
      break;
    case POP:
      break;
    case PUSH:
      break;
    case PUT:
      break;
    case GET:
      break;
    case SWAP:
      break;
    case INCREMENT:
      break;
    case DECREMENT:
      break;
    case READ:
      break;
    case UNTIL:
    case WHILE:
    case WHILENOT:
      break;
    case JUMP:
      // validate all jump targets... is target undefined, in
      // range of program etc etc.
      if (ii->a.datatype != INT) {
        fprintf(file, "error: Non integer target for jump instruction");
      }
      break;
    case JUMPTRUE:
      if (ii->a.datatype != INT) {
        fprintf(file, "error: Non integer target for jump instruction");
      }
      break;
    case JUMPFALSE:
      if (ii->a.datatype != INT) {
        fprintf(file, "error: Non integer target for jump instruction");
      }
      break;
    case TESTIS:
      if (ii->a.datatype != TEXT) {
        fprintf(file,
          "wrong datatype for parameter for TESTIS \n");
      }
      break;
    case TESTBEGINS:
      if (ii->a.datatype != TEXT) {
        fprintf(file,
          "wrong datatype for parameter for TESTBEGINS \n");
      }
      break;
    case TESTENDS:
      if (ii->a.datatype != TEXT) {
        fprintf(file,
          "wrong datatype for parameter for TESTENDS \n");
      }
      break;
    case TESTEOF:
      break;
    case TESTTAPE:
      break;
    case COUNT:
      break;
    case INCC:
      break;
    case DECC:
      break;
    case ZERO:
      break;
    case CHARS:
      break;
    case STATE:
      break;
    case QUIT:
      break;
    case WRITE:
      break;
    case NOP:
      break;
    case UNDEFINED:
      break;
    default:
      break;
  } // switch

  return -1;
} // checkInstruction

/*
check whether an instruction has the right sort of
parameters etc. It is better to do this once before a program
is executed rather than in the "execute" routine. Returns
positive integer if the instruction is not valid

s should be "validateProgram" in order to check jump
targets etc. And also to make sure jumptrue, jumpfalse have a
test before them. check for infinite loops. Check if there is
a branch zero at the end. Check if there is a read at the begining

*/

int validateProgram(struct Program * program) {
  struct Instruction * instruction;
  int ii;
  int warnings = 0; int errors = 0;

  for (ii = 0; ii < program->count; ii++) {
    instruction = &program->listing[ii];

    if (ii == program->count-1) {
      if ((instruction->command != JUMP) ||
          (instruction->a.datatype != INT) ||
          (instruction->a.number != 0)) {
        numberedInstruction(instruction, ii, stdout);
        fprintf(stdout,
          "Normally the last instruction of the program \n"
          "should be an unconditional jump to the first instruction \n"
          "eg: jump 0 \n"
          "This is because the parse machine is designed to run in a \n"
          "loop for each character read (somewhat similar to sed, except \n"
          "for characters, not lines \n");
        warnings++;
      }
    }

    if (ii == 0) {
      if (instruction->command != READ) {
        numberedInstruction(instruction, ii, stdout);
        fprintf(stdout,
          "Normally the first instruction of the program \n"
          "should be a READ, which reads one character from the \n"
          "input source \n");
        warnings++;
      }
    }

  } // for loop

  printf(
    "Checked %s%d%s instructions and found %s%d%s errors and %s%d%s warnings \n",
    BLUE, ii, NORMAL, YELLOW, errors, NORMAL, YELLOW, warnings, NORMAL);
  return 0;
}


// the virtual machine which parses
struct Machine {
  FILE * inputstream;   //source of characters
  int peep;             // next char in the stream, may have EOF
  struct Buffer buffer; //workspace & stack
  struct Tape tape;
  int accumulator;      // used for counting
  long charsRead;       // how many characters read from input stream
  long lines;           // how many lines already read from input stream
  enum Bool flag;     // used for tests
  char delimiter;     // eg *, to separate tokens on the stack
  char escape;              // escape character
  struct Program program;  // compiled instructions + ip counter
  char version[64];        // machine version string (eg: "0.1 campania" etc)
};


// all functions relating to the machine

// initialise the machine with an input stream
void newMachine(struct Machine * machine, FILE * input,
               int tapeCells, int cellSize) {
  // set the input stream
  // read the first character into peep??
  machine->inputstream = input;
  machine->charsRead = 0;
  machine->lines = 0;
  machine->accumulator = 0;
  newTape(&machine->tape, tapeCells, cellSize);
  newBuffer(&machine->buffer, 40);
  newProgram(&machine->program, 1000);
  machine->peep = fgetc(machine->inputstream);
  machine->flag = FALSE;
  machine->escape = '\\';
  machine->delimiter = '*';
  strcpy(machine->version, "0.1 campania");
}

// reset the machine without destroying program
void resetMachine(struct Machine * machine) {
  // rewind the input stream
  rewind(machine->inputstream);
  machine->charsRead = 0;
  machine->lines = 0;
  machine->accumulator = 0;
  clearTape(&machine->tape);
  clearBuffer(&machine->buffer);
  // read the first character into peep
  machine->peep = fgetc(machine->inputstream);
  machine->flag = FALSE;
  machine->escape = '\\';
  machine->delimiter = '*';
```

```c
  machine->program.ip = 0;
}

// free all memory associated with the machine.
void freeMachine(struct Machine * mm) {
  freeTape(&mm->tape);
  freeProgram(&mm->program);
  free(mm->buffer.stack);
  return;
}

// read one character from the input stream and update
// the peep and workspace and other machine registers
// this fuction is used in commands while, whilenot, until, read etc
// returns 0 if end of stream is reached
int readc(struct Machine * mm) {
   if (mm->peep == EOF) return 0;
   // not the problem
   if (feof(mm->inputstream)) return 0;
   if (ferror(mm->inputstream)) return 0;

   char * lastc = mm->buffer.workspace + strlen(mm->buffer.workspace);
   if (strlen(mm->buffer.stack) >= mm->buffer.capacity)
     growBuffer(&mm->buffer, 50);
   *lastc = mm->peep; *(lastc + 1) = '\0';

   mm->peep = fgetc(mm->inputstream);
   mm->charsRead++;
   // start line counting at 1, because that is what is expected
   if (mm->lines == 0) { mm->lines = 1; }
   if (mm->peep == '\n') { mm->lines++; }
   return 1;
}

/* display some meta-information about the machine such as
   version, capacites etc
*/
void printMachineMeta(struct Machine * machine) {
  char Colour[30];
  char date[30];
  char time[30];
  strcpy(Colour, BROWN);
  strcpy(date, "?");
  strcpy(time, "?");
  printf("%s        Machine Version:%s %s \n",
    Colour, NORMAL, machine->version);
  printf("%s         Character read:%s %ld \n",
    Colour, NORMAL, machine->charsRead);
  printf("%s       Escape character:%s %c \n",
    Colour, NORMAL, machine->escape);

  printf("%s         Peep character: %c %s \n",
    Colour, machine->peep, NORMAL);
  // cant really use escapeSpecial here, because it converts text, not
  // one character...
  // escapeSpecial(, PINK);
  printf("\n");
  printf("%s  Stack token delimiter:%s %c \n",
    Colour, NORMAL, machine->delimiter);

  /*
  printf("%s                   Flag:%s %d \n",
    Colour, NORMAL, machine->flag);
  if (machine->flag == FALSE) {
    //strcpy(date, ctime(&machine->compileDate));
  }
  */
}

void printBufferAndPeep(struct Machine * mm) {
  printf("[%ld] Buff:%s Peep:%c \n",
    mm->buffer.capacity, mm->buffer.stack, mm->peep );
}

void showBufferAndPeep(struct Machine * mm) {
  char peep[20];
  if (mm->peep == EOF) { strcpy(peep, "EOF"); }
  else if (mm->peep == '\n') { strcpy(peep, "\\n"); }
  else if (mm->peep == '\r') { strcpy(peep, "\\r"); }
  else {
    peep[0] = mm->peep; peep[1] = '\0';
  }
  printf("[%s%ld%s] Buff[%s%s%s] P[%s%s%s] \n",
    GREEN, mm->buffer.capacity, NORMAL,
    YELLOW, mm->buffer.stack, NORMAL, BLUE, peep, NORMAL );
}

/* show core registers of the parsing machine: the stack, the "workspace"
  (like a text accumulator), and the "peep" (the next character in the
  input stream */

void showStackWorkPeep(struct Machine * mm, enum Bool escape) {
  char peep[20];
  if (mm->peep == EOF) { strcpy(peep, "EOF"); }
  else if (mm->peep == '\n') { strcpy(peep, "\\n"); }
  else if (mm->peep == '\r') { strcpy(peep, "\\r"); }
  else {
    peep[0] = mm->peep; peep[1] = '\0';
  }

  // uses the %.*s, len, buffer trick to print the stack
  int stackwidth = mm->buffer.workspace - mm->buffer.stack;
  if (escape == TRUE) {

    printf("%s(Buff:%s%ld/%ld%s)%s Stack[%s%.*s%s] Work[%s",
      WHITE, BROWN,
      strlen(mm->buffer.stack), mm->buffer.capacity,
      WHITE, NORMAL,
      YELLOW, stackwidth, mm->buffer.stack, NORMAL, PURPLE);

    escapeSpecial(mm->buffer.workspace, CYAN);
    printf("%s] Peep[%s%s%s] \n",
      NORMAL, BLUE, peep, NORMAL );
  } else {
    // uses the %.*s, len, buffer trick to print the stack
    printf("%s(Buff:%s%ld%s)%s Stack[%s%.*s%s] Work[%s%s%s] Peep[%s%s%s] \n",
      WHITE, BROWN, mm->buffer.capacity, WHITE, NORMAL,
      YELLOW, stackwidth, mm->buffer.stack, NORMAL,
      PURPLE, mm->buffer.workspace, NORMAL,
      BLUE, peep, NORMAL );
  }
}

/*
 display the state of registers of the machine using colours
 This does not show the tape (which is part of the machine) or
 the program which is also part of the machine. See
 showMachineTapeProgram() or showMachineWithTape() for that.

 The "escape" parameter determines if whitespace (newlines,
 carriage returns, tabs etc) will be displayed in the format
 "\n \r \t" etc, or just printed normally.  If there are newlines
 in the workspace then the display gets messy. So its just a matter of
 aesthetics.
*/
void showMachine(struct Machine * mm, enum Bool escape) {
  showStackWorkPeep(mm, escape);
  printf("Acc:%s%d%s Flag:%s%s%s Esc:%s%c%s "
         "Delim:%s%c%s Chars:%s%ld%s Lines:%s%ld%s  \n",
    GREEN, mm->accumulator, NORMAL,
    YELLOW, (mm->flag==0?"TRUE":"FALSE"), NORMAL,
    YELLOW, mm->escape, NORMAL,
    YELLOW, mm->delimiter, NORMAL,
    BLUE, mm->charsRead, NORMAL,
    BLUE, mm->lines, NORMAL);
}

// show state of machine buffers with tape cells
void showMachineWithTape(struct Machine * mm) {
  printf("%s--------- Machine State -----------%s \n",
    BROWN, NORMAL);
  showMachine(mm, TRUE);
  printf("%s--------- Tape --------------------%s \n",
    BROWN, NORMAL);
  // true means: escape special character (newlines mainly)
  printSomeTapeInfo(&mm->tape, TRUE, 2);
}

// show state of machine buffers with tape cells
void showMachineTapeProgram(struct Machine * mm, int tapeContext) {
  printSomeProgram(&mm->program, 3);
  printf("%s--------- Machine State -----------%s \n",
    PURPLE, NORMAL);
  showMachine(mm, TRUE);
  printf("%s--------- Tape --------------------%s \n",
    PURPLE, NORMAL);
  printSomeTapeInfo(&mm->tape, TRUE, tapeContext);
}

enum ExitCode execute(struct Machine * mm, struct Instruction * ii) {

  struct TapeCell * thisCell;
  long newCapacity;
  FILE * saveFile;  // where workspace is written by 'write' command
  char * temp;      // a temporary string for x swaps
  char acc[100];    // a text version of the accumulator
  char * buffer;    // store the workspace when escaping (needs to be malloc)
  size_t len;
  int count;       // count escapable chars for malloc
  char * lastc;    // points to last char in workspace
  char * lastw;    // points to last char in workspace
  int (* fn)(int); // a function pointer for the ctype.h functions
  size_t cellLength;  // how long tapecell text is

  switch (ii->command) {
    case ADD:
      if (strlen(mm->buffer.stack) + strlen(ii->a.text) >
        mm->buffer.capacity) {
        growBuffer(&mm->buffer, strlen(ii->a.text) + 50);
      }
      strcat(mm->buffer.workspace, ii->a.text);
      break;
    case CLIP:
      if (*mm->buffer.workspace == 0) break;
      mm->buffer.workspace[strlen(mm->buffer.workspace)-1] = '\0';
      break;
    case CLOP:
      if (*mm->buffer.workspace == 0) break;
      len = strlen(mm->buffer.workspace);
      memmove(mm->buffer.workspace, mm->buffer.workspace+1, len-1);
      mm->buffer.workspace[len-1] = 0;
      break;
    case CLEAR:
      mm->buffer.workspace[0] = '\0';
      break;
    case REPLACE:
      return UNIMPLEMENTED;
      break;
    case PRINT:
      printf("%s", mm->buffer.workspace);
      break;
    case POP:
      // pop a token from the stack, so skip the first delim * and
      // read back to the next delim * in the stack buffer.
      //
      // this pop routine seems unnecessary complicated.
      // basically given s:a*b* w:
      // pop should give s:a* w:b*
      //
      if (mm->buffer.workspace == mm->buffer.stack) break;
      mm->buffer.workspace--;
      if (mm->buffer.workspace == mm->buffer.stack) {
        if (mm->tape.currentCell > 0)
          mm->tape.currentCell--;
        break;
      }
      while ((*(mm->buffer.workspace-1) != mm->delimiter) &&
             (mm->buffer.workspace-1 != mm->buffer.stack))
        mm->buffer.workspace--;

      if (mm->buffer.workspace == mm->buffer.stack) {
        if (mm->tape.currentCell > 0)
          mm->tape.currentCell--;
        break;
      }
      if (*(mm->buffer.workspace-1) != mm->delimiter)
        mm->buffer.workspace--;

      // dec current tape cell
      if (mm->tape.currentCell > 0)
        mm->tape.currentCell--;
      break;
    case PUSH:
      // could use strchr instead, or better strchrnul
      // but strchrnul is not standard C.
      if (mm->buffer.workspace[0] == '\0') break;
      while ((*mm->buffer.workspace != '\0') &&
             (*mm->buffer.workspace != mm->delimiter))
         mm->buffer.workspace++;
      if (mm->buffer.workspace[0] == mm->delimiter)
        mm->buffer.workspace++;
      // increment current tape cell
      // star at end of token - not beginning
      // look for limits !!
      if (mm->tape.currentCell < mm->tape.capacity)
        mm->tape.currentCell++;
      else {
        printf("Push: Out of tape bounds");
        exit(1);
      }
      break;
    case PUT:
      // I could make this a function, but the only place the
      // tape cell can get resized is here in the PUT command

      // if not enough space in tape cell, malloc here
      thisCell = &mm->tape.cells[mm->tape.currentCell];
      if (strlen(mm->buffer.workspace) > thisCell->capacity) {
        newCapacity = strlen(mm->buffer.workspace)+100;
        thisCell->text = malloc(newCapacity * sizeof(char));
        if (thisCell->text == NULL) {
          fprintf(stderr,
            "PUT: couldnt allocate memory for cell->text (execute) \n");
          exit(EXIT_FAILURE);
        }
        thisCell->capacity = newCapacity - 1;
        thisCell->resizings++;
      }
      strcpy(thisCell->text, mm->buffer.workspace);
      break;
    case GET:
      cellLength = strlen(mm->tape.cells[mm->tape.currentCell].text);
      if ((strlen(mm->buffer.stack) + cellLength) > mm->buffer.capacity) {
        growBuffer(&mm->buffer, cellLength + 100);
      }
      strcat(mm->buffer.workspace,
        mm->tape.cells[mm->tape.currentCell].text);
      break;
    case SWAP:
      temp = malloc(strlen(mm->buffer.workspace + 10) * sizeof(char));
      // ... todo! implement this
      strcpy(temp, mm->buffer.workspace);
      cellLength = strlen(mm->tape.cells[mm->tape.currentCell].text);
      // this is a bug, we need a function mm.workspaceCapacity()
      // because some room in the buffer is taken up by the stack
      // and we dont know how long that is because the stack is
      // not zero terminated (it is combined with the workspace).
      if (cellLength > workspaceCapacity(&mm->buffer)) {
        growBuffer(&mm->buffer, cellLength + 20);
      }
      strcpy(mm->buffer.workspace,
        mm->tape.cells[mm->tape.currentCell].text);

      thisCell = &mm->tape.cells[mm->tape.currentCell];
      if (strlen(temp) > thisCell->capacity) {
        newCapacity = strlen(temp)+20;
        thisCell->text = malloc(newCapacity * sizeof(char));
        if (thisCell->text == NULL) {
          fprintf(stderr,
            "PUT: couldnt allocate memory for cell->text (execute) \n");
          exit(EXIT_FAILURE);
        }
        thisCell->capacity = newCapacity - 1;
        thisCell->resizings++;
      }
      strcpy(thisCell->text, temp);
      free(temp);
      break;
    case INCREMENT:
      if (mm->tape.currentCell < mm->tape.capacity)
        mm->tape.currentCell++;
      else {
        printf("++: Out of tape bounds");
        exit(1);
      }
      break;
    case DECREMENT:
      if (mm->tape.currentCell > 0)
        mm->tape.currentCell--;
      break;
    case READ:
      if (mm->peep == EOF) {
        return ENDOFSTREAM;   // end of file
      }
      readc(mm);
      break;
    case UNTIL:
      // until workspace ends with ii->a.text
      // this seems to be working.
      if (mm->peep == EOF) break;
      if (!readc(mm)) break;

      len = strlen(ii->a.text);
      size_t worklen = strlen(mm->buffer.workspace);
      // below is a general "endswith" function...
      // strcmp(mm->buffer.workspace+worklen-len, ii->a.text) != 0) {

      char * suffix = mm->buffer.workspace+worklen-len;
      //while (strcmp(mm->buffer.workspace+worklen-len, ii->a.text) != 0) {
      while (readc(mm)) {
        // zero false, non-zero true. So if strings are equal stop looping
        // and no escape of first char in suffix
        worklen++;
        suffix = mm->buffer.workspace+worklen-len;
        // if ((strcmp(suffix, ii->a.text) == 0) && (*(suffix-1) != '\\')) {
        // deal with escape character.
        if ((strcmp(suffix, ii->a.text) == 0) && (*(suffix-1) != mm->escape)) {
          // printf("%s\n", suffix);
          break;
        }
      }
      break;
    case WHILE:
      if (mm->peep == EOF) break;
      // while and whilenot handle classes eg :space: ranges eg [a-z]
      // and lists eg [abxy] [.] etc
      // a function pointer: fn = &isblank; int res = (*fn)('1');

      if (ii->a.datatype == CLASS) {
        // get character class function pointer from the instruction
        fn = ii->a.classFn;
        while ((*fn)(mm->peep)) {
          if (!readc(mm)) break;
        }
      }
      else if (ii->a.datatype == RANGE) {
        // compare peep to a range of characters eg a-z
        while ((mm->peep >= ii->a.range[0]) && (mm->peep <= ii->a.range[1])) {
          if (!readc(mm)) break;
        }
      }
      else if (ii->a.datatype == LIST) {
        // read input while peep is in a list of chars
        while (strchr(ii->a.list, mm->peep) != NULL) {
          if (!readc(mm)) break;
        }
      }
      break;
    case WHILENOT:
      // why not use a switch here??? Its a switch within a switch...
      /*
       switch (ii->a.datatype) {
         case CLASS:
            break;
         case RANGE:
            break;
       }
      */

      if (ii->a.datatype == CLASS) {
        // get character class function pointer from the instruction
        fn = ii->a.classFn;
        while (!(*fn)(mm->peep)) {
          if (!readc(mm)) break;
        }

      }
      else if (ii->a.datatype == RANGE) {
        // read input while peep is not in a range (eg b-f)
        while ((mm->peep < ii->a.range[0]) || (mm->peep > ii->a.range[1])) {
          if (!readc(mm)) break;
        }
      }
      else if (ii->a.datatype == LIST) {
        // read input while peep is not in a char list eg "abxy"
        while (strchr(ii->a.list, mm->peep) == NULL) {
          if (!readc(mm)) break;
        }
      }
      break;
    case JUMP:
      // update program counter. non-relative jump
      mm->program.ip = ii->a.number;
      break;
    case JUMPTRUE:
      if (mm->flag == TRUE)
        // relative jump. easier to assemble code
        // mm->program.ip = ii->a.number;
        mm->program.ip = mm->program.ip + ii->a.number;
      else mm->program.ip++;
      break;
    case JUMPFALSE:
      if (mm->flag == FALSE)
        // relative jump.
        mm->program.ip = mm->program.ip + ii->a.number;
      else mm->program.ip++;
      break;
    case TESTIS:
      if (strcmp(mm->buffer.workspace, ii->a.text) == 0) {
        mm->flag = TRUE;
      } else { mm->flag = FALSE; }
      break;
    case TESTCLASS:
      // handle class, range, text, etc
      // depending on the parameter type.
      if (ii->a.datatype == CLASS) {
        // get character class function pointer from the instruction
        fn = ii->a.classFn;
        lastc = mm->buffer.workspace;
        //while ((*fn)(mm->peep)) {
        mm->flag = TRUE;
        while (*lastc != 0) {
          if (!fn(*lastc)) {
            mm->flag = FALSE; break;
          }
          lastc++;
        }
      }
      else if (ii->a.datatype == RANGE) {
        // compare ws to a range of characters eg a-z
        mm->flag = TRUE;
        lastc = mm->buffer.workspace;
        while (*lastc != 0) {
          if ((*lastc <= ii->a.range[0]) || (*lastc >= ii->a.range[1])) {
            mm->flag = FALSE; break;
          }
          lastc++;
        }
      }
      else if (ii->a.datatype == LIST) {
        // compare ws to a list of chars
        //while (strchr(ii->a.list, mm->peep) != NULL) {
        mm->flag = TRUE;
        lastc = mm->buffer.workspace;
        while (*lastc != 0) {
          if (strchr(ii->a.list, *lastc) == NULL) {
            mm->flag = FALSE; break;
          }
          lastc++;
        }
      }
      break;
    case TESTBEGINS:
      if (strncmp(mm->buffer.workspace, ii->a.text, strlen(ii->a.text)) == 0) {
        mm->flag = TRUE;
      } else { mm->flag = FALSE; }
      break;
    case TESTENDS:
      if (strcmp(mm->buffer.workspace + strlen(mm->buffer.workspace)
           - strlen(ii->a.text), ii->a.text) == 0)
        mm->flag = TRUE;
      else mm->flag = FALSE;
      break;
    case TESTEOF:
      if (mm->peep == EOF) { mm->flag = TRUE; }
      else { mm->flag = FALSE; }
      break;
    case TESTTAPE:
      if (strcmp(mm->buffer.workspace,
        mm->tape.cells[mm->tape.currentCell].text) == 0)
        { mm->flag = TRUE; }
      else { mm->flag = FALSE; }
      break;
    case COUNT:
      sprintf(acc, "%d", mm->accumulator);
      if (strlen(mm->buffer.stack) + strlen(acc) > mm->buffer.capacity) {
        growBuffer(&mm->buffer, strlen(acc) + 50);
      }
      strcat(mm->buffer.workspace, acc);
      break;
    case INCC:
      mm->accumulator++;
      break;
    case DECC:
      mm->accumulator--;
      break;
    case ZERO:
      mm->accumulator = 0;
      break;
    case CHARS:
      sprintf(acc, "%ld", mm->charsRead);
      if (strlen(mm->buffer.stack) + strlen(acc) > mm->buffer.capacity) {
        growBuffer(&mm->buffer, strlen(acc) + 50);
      }
      strcat(mm->buffer.workspace, acc);
      break;
    case LINES:
      sprintf(acc, "%ld", mm->lines);
      if (strlen(mm->buffer.stack) + strlen(acc) > mm->buffer.capacity) {
        growBuffer(&mm->buffer, strlen(acc) + 50);
      }
      strcat(mm->buffer.workspace, acc);
      break;
    case ESCAPE:
      // count escapable
      // but if the escapable is already preceded with a
      // backslash should we reescape it
      count=0;
      lastc = strchr(mm->buffer.workspace, ii->a.text[0]);
      while (lastc != NULL) {
        count++;
        lastc = strchr(lastc+1, ii->a.text[0]);
      }
      buffer = malloc((strlen(mm->buffer.workspace)+count+10) * sizeof(char));
      // also grow workspace if needed.
      strcpy(buffer, mm->buffer.workspace);
      lastw = mm->buffer.workspace;
      while (*buffer != 0) {
        if (*buffer == ii->a.text[0]) {
          *lastw = mm->escape;
          lastw++;
        }
        *lastw = *buffer;
        buffer++; lastw++;
      }
      *lastw = 0;
      break;
    case UNESCAPE:
      /* how should this work? should unescape deescape all
         backlash letters or only one or a list, or a
         class :blank: or a range [a-z] */
      lastc = mm->buffer.workspace;
      lastw = mm->buffer.workspace;
      while (*lastc != 0) {
        if ((*lastc == mm->escape) && (*(lastc+1)==ii->a.text[0])) {
          lastc++;
        }
        *lastw = *lastc;
        lastc++; lastw++;
      }
      *lastw = 0;
      // ...
      break;
    case STATE:
      showMachineTapeProgram(mm, 3);
      break;
    case QUIT:  //
      return EXECQUIT;  // script must now exit
      break;
    case BAIL:  //
      return BADQUIT;  // script must now exit with error code
      break;
    case WRITE:
      // write workspace to file 'sav.pp'
      if ((saveFile = fopen("sav.pp", "w")) == NULL) {
        printf ("Cannot open file %s'sav.pp'%s for writing\n",
            YELLOW, NORMAL);
        return WRITESAVERROR;
      }
      fputs(mm->buffer.workspace, saveFile);
      fclose(saveFile);
      break;
    case NOP:
      break;
    case UNDEFINED:
      fprintf(stderr,
           "Executing undefined command! "
           "at instruction: %d \n ", mm->program.ip);
      return EXECUNDEFINED;  // error code
      break;
  }
  // if a jump command, dont increment the instruction pointer
  if (strncmp(info[ii->command].name, "jump", 4) != 0) {
    mm->program.ip++;
  }
  return SUCCESS;
}


// steps through one instruction of the machines program
void step(struct Machine * mm) {
  //int result =
  execute(mm, &mm->program.listing[mm->program.ip]);
}


/*
 runs the compiled program in the machine
 but this will exit when the last read is performed...
 execute() has the following exit codes which we might need to
 handle:
  0: success no problems
  1: end of stream reached (tried to read eof)
  2: trying to execute undefined instruction
  3: quit/crash command executed (exit script)

*/

enum ExitCode run(struct Machine * mm) {
  int result;
  for (;;) {
    result = execute(mm, &mm->program.listing[mm->program.ip]);
    if (result != 0) break;
  }
  return result;
}

void runDebug(struct Machine * mm) {
  int result;
  long ii = 0;  // a counter
  for (;;) {
    printf("%6ld: ip=%3d T(n)=%3d", ii, mm->program.ip, mm->tape.currentCell);
    printInstruction(&mm->program.listing[mm->program.ip]);
    result = execute(mm, &mm->program.listing[mm->program.ip]);
    printf("\n");
    if (result != 0) {
      printf("execute() returned error code (%d)\n", result);
      break;
    }
    ii++;
  }
}

// runs the compiled program in the machine until the
// flag register is set to true
void runUntilTrue(struct Machine * mm) {
  int result;
  while (mm->flag == FALSE) {
    result = execute(mm, &mm->program.listing[mm->program.ip]);
    if (result != 0) break;
  }
}

/*
  runs the compiled program in the machine until the workspace
  is exactly the specified text */

void runUntilWorkspaceIs(struct Machine * mm, char * text) {
  int result;
  while ((mm->peep != EOF) && (strcmp(mm->buffer.workspace, text) != 0)) {
    result = execute(mm, &mm->program.listing[mm->program.ip]);
    if (result != 0) break;
  }
}

int endsWith(const char *str, const char *suffix)
{
  if (!str || !suffix) return 0;
  size_t lenstr = strlen(str);
  size_t lensuffix = strlen(suffix);
  if (lensuffix >  lenstr) return 0;
  return strncmp(str + lenstr - lensuffix, suffix, lensuffix) == 0;
}

/*
  runs the compiled program until the workspace
  is ends with the specified text */
void runUntilWorkspaceEndsWith(struct Machine * mm, char * text) {
  int result;
  while ((mm->peep != EOF) && (!endsWith(mm->buffer.workspace, text))) {
    result = execute(mm, &mm->program.listing[mm->program.ip]);
    if (result != 0) break;
  }
}

// given some instruction text (eg: add "this") compile an instruction into
// the machines program listing. Returns zero on success or a positive integer
// if arguments are invalid.  maybe this should return a pointer to an
// instruction upon success the job of compile is to housekeep the machine.
// the job of instructionFromText() is actually to parse the text into an
// instruction
// change this to struct Program * pp ?? or change
// loadAssembledProgram to take a machine

int compile(struct Program * program, char * text,
            int pos, struct Label table[]) {
  struct Instruction * ii;
  char command[200];
  char args[1000];
  // pos is where in the program list to put the compiled instruction
  ii = &program->listing[pos];

  if (program->capacity == program->count + 1) {
    growProgram(program, 10);
  }

  // int result =
  sscanf(text, "%200s %200[^\n]", command, args);

  enum Command com = textToCommand(command);
  if (com == UNDEFINED) {
    printf("Unknown command name %s%s%s \n", BROWN, command, NORMAL);
    printf("on line: %s%s%s at program position %d \n",
      BROWN, text, NORMAL, pos);
    return 1;
  }
  instructionFromText(stdout, ii, text, pos, table);
  program->count++;

  return 0;
  // 2 argument compilation... to do
} // compile
// given user input text return the test command
enum TestCommand textToTestCommand(const char * text) {
  int ii;
  if (*text == 0) return UNKNOWN;
  for (ii = 0; ii < UNKNOWN; ii++) {
    if ((strcmp(text, testInfo[ii].names[0]) == 0) ||
        (strcmp(text, testInfo[ii].names[1]) == 0)) {
       return (enum TestCommand)ii;
    }
  }
  return UNKNOWN;
}

void showTestHelp() {
  int ii;
  int key;
  printf("%s[ALL HELP COMMANDS]%s:\n", YELLOW, NORMAL);
  for (ii = 0; ii < UNKNOWN; ii++) {
    printf("%s%5s%s %s %s-%s %s%s \n",
      GREEN, testInfo[ii].names[0], PALEGREEN, testInfo[ii].argText,
      NORMAL, WHITE, testInfo[ii].description, NORMAL);
    if ( (ii+1) % 14 == 0 ) {
      pagePrompt();
      key = getc(stdin);
      if (key == 'q') { return; }
    }
  }
}

// information about different modes the test program can be in
// eg step where enter steps through the next instruction.
// Or maybe 2 mode variables enterMode and programMode. enterMode
// determines what the <enter> key does, and programMode determines
// whether instructions are compiled or executed or both
enum TestMode {
  COMPILE=0, IPCOMPILE, ENDCOMPILE, INTERPRET,
  MACHINE, PROGRAM, STEP, RUN } mode;
// contain information about all commands
struct {
  enum TestMode mode;
  char * name;
  char * description;
} modeInfo[] = {
  { COMPILE, "compile",
    "enter displays the compiled program. Instructions entered are \n"
    "compiled into the machines program listing, but are not executed "},
  { IPCOMPILE, "ipcompile",
    "Instructions are compiled into the machines program listing \n"
    "at the current instruction pointer position instead of at  \n"
    "the end of the program. This is a slightly clunky way of \n"
    "modifying the in-memory program, but possibly easier than trying \n"
    "to manually modify the text assembler listing. "},
  { ENDCOMPILE, "endcompile",
    "Instructions are compiled into the machines program listing \n"
    "at the end of the program  \n" },
  { INTERPRET, "interpret",
    "Instructions entered are executed but not compiled into the \n"
    "machine's program.  \n" },
  { MACHINE, "machine", "enter displays the state of the machine"},
  { PROGRAM, "", ""},
  { STEP, "step",
    "When enter is pressed the next instruction is executed and "
    "the state of the machine is displayed "},
  { RUN, "run",
    "When enter is pressed the current program is run. "}
};

// searches testInfo array for commands matching a search term
void searchHelp(char * text) {
  int ii; int jj = 0;
  char key;
  printf("%s[Searching help commands]%s:\n", YELLOW, NORMAL);
  for (ii = 0; ii < UNKNOWN; ii++) {
    if ((strstr(testInfo[ii].description, text) != NULL) ||
        (strstr(testInfo[ii].names[1], text) != NULL) ||
        (strstr(testInfo[ii].names[0], text) != NULL))  {
      jj++;
      printf("%s%5s%s %s %s-%s %s%s \n",
        PURPLE, testInfo[ii].names[0], YELLOW, testInfo[ii].argText,
        NORMAL, BROWN, testInfo[ii].description, NORMAL);
      if ( (jj+1) % 14 == 0 ) {
        pagePrompt();
        key = getc(stdin);
        if (key == 'q') { return; }
      }
    } // if search term found
  } // for

  if (jj == 0) {
    printf("No results found for '%s'\n", text);
  }
}

/* load a script from file and install in the machines program at
   instruction number "position". This allows us to append one
   script at the end of another which may be useful. */
enum ExitCode loadScript(
   struct Program * program, FILE * scriptFile, int position) {

    /* the procedure is:
      load asm.pp, run it on the script-file (assembly is saved to sav.pp)
      load sav.pp, run it on the input stream. */
    FILE * asmFile;
    if ((asmFile = fopen("asm.pp", "r")) == NULL) {
      printf("Could not open assembler %sasm.pp%s \n",
        YELLOW, NORMAL);
      return 1;
    }
    FILE * savFile;
    // first delete contents of the sav.pp file to avoid confusion
    // later.
    if ((savFile = fopen("sav.pp", "w")) == NULL) {
      printf("Could not open script file %s'sav.pp'%s for writing \n",
        YELLOW, NORMAL);
      return(1);
    }

    fputs("add \"no script\" \n", savFile);
    fputs("quit \n", savFile);
    fclose(savFile);

    struct Machine new;
    newMachine(&new, scriptFile, 100, 10);
    loadAssembledProgram(&new.program, asmFile, 0);

    int result = 0;
    result = run(&new);
    // printf("result of run(): "); printExitCode(result); exit(1);
    //runDebug(&m);
    fclose(asmFile);
    freeMachine(&new);

    /*
     check if the compilation was successful. (which means
     ExitCode SUCCESS of EXECQUIT) If not successful, do not proceed.
     because the script file was not properly compiled by asm.pp
    */
    if (result > EXECQUIT) return result;

    // asm.pp has created a new sav.pp file, which is the
    // script in a "compiled" form (a type of assembly language
    // for the parse virtual machine. We can now open, load
    // it and run it.
    if ((savFile = fopen("sav.pp", "r")) == NULL) {
      printf("Could not open script file %s'sav.pp'%s for reading. \n",
        YELLOW, NORMAL);
      return READSAVERROR;
    }
    loadAssembledProgram(program, savFile, 0);
    return SUCCESS;
}

void printUsageHelp() {
  fprintf(stdout, "'pp' a pattern parsing machine \n"
    "Usage: pp [-shI] [-f script-file] [-a file] [inputfile] \n"
    " \n"
    "  -f script-file   text input file \n"
    "  -e expression    add inline script commands to script \n"
    "  -h               print this help \n"
    "  -s               run in unix filter mode (the default).\n"
    "  -a [file]        load script-assembly source file \n"
    "  -I               run in interactive mode (with shell) \n");
    //"  -M               print the state of the machine after compiling \n"
    //"                   a script. This option is useful for debugging. \n"
```

```
}

// The main testing loop for the machine. This accepts interactive
// commands and executes them, among other things. Allows the state
// of the machine to be observed, including the compiled program.
int main (int argc, char *argv[]) {

  int c;
  // name of the file with script assembly commands
  char * asmFileName = NULL;
  // name of file to serve as input stream
  char * inputFile = NULL;
  // name of the file with parse script commands
  char * scriptFileName = NULL;
  // scripts commands on the command line with the -e switch
  char * inlineScript = NULL;
  char source[64] = "input.txt";
  // if asm.pp generate an error, then the machine parse state will
  // be printed.
  enum Bool printState = FALSE;
  /* filtermode means the program is used as a unix filter, like sed, grep etc
     intermode means the program starts an interactive shell, so that
     the user can debug scripts and learn about the machine */
  enum { FILTERMODE, INTERMODE } progmode = FILTERMODE;
  // determines how much of the tape will be shown by printSomeTapeInfo()
  // and showMachineTapeProgram()
  int tapeContext = 3;
  /*
  //A mystery, this takes input from a pipe in test.stdin.c but
  //not here, only looks at console

    char word[128];
    FILE * fp = stdin;
    while (fscanf(fp, "%127s", word) == 1) {
       printf("%s\n", word);
    }
    exit(0);
   */

  opterr = 0;

  while ((c = getopt (argc, argv, "i:f:e:IsMha:")) != -1) {
    switch (c) {
      // a text input file, but this should be a non-option argument
      // (just the file name).
      case 'f':
        scriptFileName = optarg;
        break;
      case 'e':
        inlineScript = optarg;
        break;
      case 'I':
        progmode = INTERMODE;
        break;
      case 's':
        progmode = FILTERMODE;
        break;
      case 'M':
        printState = TRUE;
        break;
      case 'h':
        printUsageHelp();
        break;
      case 'a':
        asmFileName = optarg;
        break;
      case '?':
        switch (optopt) {
          case 'a':
```

```
          fprintf (stderr, "Option -%c requires an argument.\n", optopt);
          fprintf (stderr, " -a scriptAssemblyFile \n");
          break;
        case 'f':
          fprintf (
            stderr, "%s: option -%c requires an argument.\n",
            argv[0], optopt);
          break;
        case 'e':
          fprintf (
            stderr, "%s: option -%c requires an argument.\n",
            argv[0], optopt);
          break;
        default:
          fprintf (stderr, "%s: unknown option -- %c \n", argv[0], optopt);
          break;
      }
      printUsageHelp();
      exit(1);
    default:
      abort ();

  }
} // while

if (argv[optind] != NULL)
  { inputFile = argv[optind]; }
else {
  fprintf(stdout,
    "no input file given. This is actually a bug because obviously \n"
    "%s should accept input from standard input.\n", argv[0]);
  printUsageHelp();
  exit(1);
}

if ((asmFileName != NULL) && (scriptFileName != NULL)) {
  printf("cannot load assembly and script at the same time (-a and -f)");
  printUsageHelp();
  exit(1);
}

if (inputFile != NULL) {
  strcpy(source, inputFile);
}

char version[64] = "v31415";

if (progmode != FILTERMODE) {
  banner() ;
  printf("Compiled: %s%s%s \n", YELLOW, version, NORMAL);
}

struct Machine m;

//struct Instruction i;

// mode determines how the enter key behaves. Also, in compile mode
// instructions are compiled into the machines program but not automatically
// executed
enum TestMode mode = INTERPRET;
enum TestCommand testCom;

FILE * inputStream;

  if((inputStream = fopen (source, "r")) == NULL) {
    printf ("Cannot open file %s%s%s \n", YELLOW, source, NORMAL);
    printf ("Try %s%s -i <inputfile>%s \n", CYAN, argv[0], NORMAL);
    exit (1);
  }
```

```
  //else { inputStream = stdin; }

if (progmode != FILTERMODE) {
  printf("Using source file %s%s%s as input stream \n",
    YELLOW, source, NORMAL);
  printUsefulCommands();

     /*
        "%s%s%s see all available commands \n"
        "%s%s%s view machine instructions \n",
        YELLOW, testInfo[HELP].names[0], NORMAL,
        YELLOW, testInfo[LISTMACHINECOMMANDS].names[0], NORMAL
      );
      */

}

FILE * asmFile;
FILE * scriptFile;

// machine, input stream, tape cells, tape cells size, and program listing
newMachine(&m, inputStream, 100, 10);
// now we can use the machine to parse etc
if (asmFileName != NULL) {
  if ((asmFile = fopen(asmFileName, "r")) == NULL) {
    printf("Could not open script-assembler file %s%s%s \n",
      YELLOW, asmFileName, NORMAL);
    exit(1);
  }
  if (progmode != FILTERMODE) {
    printf("\n");
    printf("Loading assembler file %s%s%s \n", YELLOW, asmFileName, NORMAL);
  }
  strcpy(m.program.source, "asm.pp");
  loadAssembledProgram(&m.program, asmFile, 0);

  if (progmode != FILTERMODE) {

    printf("Compiled %s%d%s instructions "
           "from '%s%s%s' in about %s%ld%s milliseconds \n",
      CYAN, m.program.count, NORMAL,
      CYAN, m.program.source, NORMAL,
      CYAN, m.program.compileTime, NORMAL);
  }
  fclose(asmFile);
}

if (scriptFileName != NULL) {
  if ((scriptFile = fopen(scriptFileName, "r")) == NULL) {
    printf("Could not open script file %s%s%s \n",
      YELLOW, scriptFileName, NORMAL);
    exit(1);
  }
  if (progmode != FILTERMODE) {
    printf("\n");
    printf("Loading script file %s%s%s \n", YELLOW, scriptFileName, NORMAL);
  }
  /* the procedure is:
    load asm.pp, run it on the script-file (assembly is saved to sav.pp)
    load sav.pp, run it on the input stream. */
  int result = loadScript(&m.program, scriptFile, 0);
  fclose(scriptFile);
  /*
  if (printState == TRUE) {
    showMachineTapeProgram(&m, 3);
  } */

  if (result > EXECQUIT) {
    fprintf(stderr,
        "The script file '%s' could not be compiled. \n", scriptFileName);
      printExitCode(result);
      exit(result);
    }
  }

  /* add commands given to the -e switch to the current program */
  if (inlineScript != NULL) {

    /* the procedure is:
      save the -e script commands to a temporary file
      compile the commands to sav.pp with asm.pp
      load sav.pp, run it on the input stream. */
    FILE * temp = NULL;
    if ((temp = fopen("temp.pp", "w")) == NULL) {
      printf("Could not open temporary file %stemp.pp%s for writing \n",
        YELLOW, NORMAL);
      exit(1);
    }

    fputs(inlineScript, temp);
    // this is a cludge because asm.pp doesnt deal with
    // the last character of input.
    fputs(" ", temp);
    fclose(temp);

    if ((temp = fopen("temp.pp", "r")) == NULL) {
      printf("Could not open temporary file %stemp.pp%s for reading \n",
        YELLOW, NORMAL);
      exit(1);
    }

    if ((asmFile = fopen("asm.pp", "r")) == NULL) {
      printf("Could not open assembler %sasm.pp%s \n",
        YELLOW, NORMAL);
      exit(1);
    }

    if (progmode != FILTERMODE) {
      printf("Loading assembler %sasm.pp%s \n", YELLOW, NORMAL);
    }

    // need a parameter to loadScript if we want to show machine
    // state after a compilation error
    int result = loadScript(&m.program, temp, m.program.count);
    fclose(temp);
    if (result > EXECQUIT) {
      fprintf(stderr,
        "The inline script in -e '%s' "
        "was not compiled. \n", inlineScript);
      printExitCode(result);
      exit(result);
    }
  }

  if (progmode == FILTERMODE) {
    //or runDebug(&m);
    run(&m);
    exit(0);
  }

  char line[401];
  char command[201];
  char argA[201];
  char argB[201];
  char args[300];

  while (1) {
    printf(">");
    fgets(line, 400, stdin);
    // remove newline
    line[strlen(line) - 1] = '\0';
    //printf("input[%s]\n", line);

    command[0] = args[0] = argA[0] = argB[0] = '\0';
    // int result;
    sscanf(line, "%200s %200[^\n]", command, args);

    // we also need to deal with quoted arguments such as
    //  a "many words"
    //  eg ranges [a-z] character classes :alpha: etc.

   /*
   A good example of sscanf with different cases
    res = sscanf(buf,
      " \"%5[^\"]\" \"%19[^\"]\" \"%29[^\"]\" %lf %d",
      p->number, p->name, p->description, &p->price, &p->qty);

   if (res < 5) {
     static const char *where[] = {
        "number", "name", "description", "price", "quantity"};
    if (res < 0) res = 0;
    fprintf(stderr,
      "Error while reading %s in line %d.\n", where[res], nline);
       break;
   }
   */

    // eg sscanf(line, "%200s '%200[^']'", command, argA)
    // printf("c=%s, argA=%s, argB=%s \n", command, argA, argB);

    testCom = textToTestCommand(command);
    //printf("testcom= %s \n", testInfo[testCom].description);

    if (testCom == HELP) {
      showTestHelp();
    }
    else if (testCom == SEARCHCOMMAND) {
      if (strlen(args) == 0) {
        printf("No command seach term specified. \n");
        printf("Try: // <search-term> \n");
        continue;
      }
      searchCommandHelp(args);
    }
    else if (testCom == SEARCHHELP) {
      if (strlen(args) == 0) {
        printf("No seach term specified. \n");
        printf("Try: / <search-term> \n");
        continue;
      }
      searchHelp(args);
    }
    else if (testCom == COMMANDHELP) {
      if (strlen(args) == 0) {
        printf("No machine command specified. \n");
        continue;
      }
      if (textToCommand(args) == UNDEFINED) {
        printf("Not a valid machine command %s%s%s \n", YELLOW, args, NORMAL);
        continue;
      }
      enum Command thisCom = textToCommand(args);
      printf("name: %s %s(%s%c%s)%s \n",
        info[thisCom].name, GREY, GREEN, info[thisCom].abbreviation,
        GREY, NORMAL);
      printf("   %s \n", info[thisCom].shortDesc);
      printf("   %s \n", info[thisCom].longDesc);
      printf("example: %s \n", info[thisCom].example);
      printf("takes %s%d%s argument(s) \n", YELLOW, info[thisCom].args, NORMAL);
    }
    else if (testCom == LISTMACHINECOMMANDS) {
      showCommandNames();
    }
    else if (testCom == DESCRIBEMACHINECOMMANDS) {
      machineCommandHelp();
    }
    else if (testCom == LISTCLASSES) {
      printClasses();
    }
    else if (testCom == LISTCOLOURS) {
      showColours();
    }
    else if (testCom == MACHINEPROGRAM) {
      showMachineTapeProgram(&m, tapeContext);
    }
    else if (testCom == MACHINESTATE) {
      showMachine(&m, TRUE);
    }
    else if (testCom == MACHINETAPESTATE) {
      showMachineWithTape(&m);
    }
    else if (testCom == MACHINEMETA) {
      printMachineMeta(&m);
    }
    else if (testCom == BUFFERSTATE) {
      showBufferInfo(&m.buffer);
    }
    else if (testCom == STACKSTATE) {
      showStackWorkPeep(&m, TRUE);
    }
    else if (testCom == TAPESTATE) {
      printSomeTape(&m.tape, TRUE);
    }
    else if (testCom == TAPEINFO) {
      printTapeInfo(&m.tape);
    }
    else if (testCom == TAPECONTEXTLESS) {
      if (tapeContext > 0) tapeContext--;
      printf("Tape display variable set to %d\n", tapeContext);
    }
    else if (testCom == TAPECONTEXTMORE) {
      tapeContext++;
      printf("Tape display variable set to %d\n", tapeContext);
    }
    else if (testCom == RESETINPUT) {
      rewind(inputStream);
      printf ("%s Rewound the input stream %s \n", YELLOW, NORMAL);
    }
    else if (testCom == RESETMACHINE) {
      colourPrint("reinitializing machine ...\n");
      resetMachine(&m);
    }
    else if (testCom == STEPMODE) {
      mode = STEP;
      printf(
        "%sSTEP%s mode (<enter> steps next instruction)\n",
        YELLOW, NORMAL);
    }
    else if (testCom == PROGRAMMODE) {
      mode = PROGRAM;
      printf(
        "%sPROGRAM%s mode (<enter> displays program listing)\n",
         YELLOW, NORMAL);
    }
    else if (testCom == MACHINEMODE) {
      mode = MACHINE;
      printf(
        "%sMACHINE%s mode (<enter> displays machine state)\n",
        YELLOW, NORMAL);
    }
    else if (testCom == COMPILEMODE) {
      printf ("Not implemented ... \n");
    }
    else if (testCom == IPCOMPILEMODE) {
      mode = IPCOMPILE;
      printf ("Mode set to %sIPCOMPILE%s \n", YELLOW, NORMAL);
      printf (
        "  Instructions will be compiled into the program \n"
        "  at the current instruction pointer position, instead \n"
        "  of at the end of the program. \n");
    }
    else if (testCom == ENDCOMPILEMODE) {
      mode = ENDCOMPILE;
      printf ("Mode set to %sENDCOMPILE%s \n", YELLOW, NORMAL);
    }
    else if (testCom == INTERPRETMODE) {
      mode = INTERPRET;
      printf("Mode set to %sINTERPRET%s \n", YELLOW, NORMAL);
      printf("  %smachine commands will be executed but not compiled \n"
             "  to the internal program.%s \n",
             WHITE, NORMAL);
    }
    // ENTER key pressed ....
    else if (strcmp(command, "") == 0) {
      if (mode == MACHINE) {
        // to do: show the stream with ftell and fseek
        showMachine(&m, TRUE);
      }
      else if (mode == STEP) {
        step(&m);
        printSomeProgram(&m.program, 5);
        showMachine(&m, TRUE);
      }
      else if (mode == PROGRAM) {
        printSomeProgram(&m.program, 5);
      }
    }
    else if (strcmp(command, "BB") == 0) {
      showBufferAndPeep(&m);
    }
    else if (testCom == LISTPROGRAM) {
      printProgram(&m.program);
    }
    else if (testCom == LISTSOMEPROGRAM) {
      printSomeProgram(&m.program, 7);
    }
    else if (testCom == LISTPROGRAMWITHLABELS) {
      printProgramWithLabels(&m.program);
    }
    else if (testCom == PROGRAMMETA) {
      printProgramMeta(&m.program);
    }
    else if (testCom == SAVEPROGRAM) {
      FILE * savefile;
      if (strlen(args) == 0) {
        printf ("%s No save file given! %s \n", CYAN, NORMAL);
        printf ("%s Try %sP.wa <filename> %s \n", GREEN, YELLOW, NORMAL);
        continue;
      }
      if ((savefile = fopen (args, "w")) == NULL) {
        printf ("Could not open file %s%s%s for writing \n",
          YELLOW, args, NORMAL);
        continue;
      }
      saveAssembledProgram(&m.program, savefile);
      fclose(savefile);
    }
    else if (testCom == SHOWJUMPTABLE) {
      printJumpTable(m.program.labelTable);
    }
    else if (testCom == LOADPROGRAM) {
      FILE * loadfile;
      if (strlen(args) == 0) {
        printf ("%s No assembly text file given to load %s \n", GREEN, NORMAL);
        continue;
      }
      if ((loadfile = fopen (args, "r")) == NULL) {
        printf ("Could not open file %s%s%s for reading\n", YELLOW, args, NORMAL);
        continue;
      }
      loadAssembledProgram(&m.program, loadfile, 0);
      strcpy(m.program.source, args);
      fclose(loadfile);
    }
    else if (testCom == LOADASM) {
      FILE * loadfile;
      if ((loadfile = fopen ("asm.pp", "r")) == NULL) {
        printf ("Could not open file %sasm.pp%s for reading\n",
           YELLOW, NORMAL);
        continue;
      }
      printf("%sresetting machine and loading '%sasm.pp%s'... \n",
        BROWN, PINK, NORMAL);
      rewind(inputStream);
      resetMachine(&m);
      clearProgram(&m.program);
      strcpy(m.program.source, "asm.pp");
      loadAssembledProgram(&m.program, loadfile, 0);
      fclose(loadfile);
    }
    else if (testCom == LOADLAST) {
      FILE * loadfile;
      if ((loadfile = fopen ("last.pp", "r")) == NULL) {
        printf ("Could not open file %slast.pp%s for reading\n",
          YELLOW, NORMAL);
        continue;
      }
      strcpy(m.program.source, "last.pp");
      loadAssembledProgram(&m.program, loadfile, 0);
      fclose(loadfile);
    }
    else if (testCom == LOADSAVED) {
      FILE * loadfile;
      if ((loadfile = fopen ("sav.pp", "r")) == NULL) {
        printf ("Could not open file %ssav.pp%s for reading\n",
          YELLOW, NORMAL);
        continue;
      }
      strcpy(m.program.source, "sav.pp");
      loadAssembledProgram(&m.program, loadfile, 0);
      fclose(loadfile);
    }
    else if (testCom == LISTSAVFILE) {
      FILE * loadfile;
      char line[200];
      char key;
      int ii = 0;

      if ((loadfile = fopen("sav.pp", "r")) == NULL) {
        printf ("Could not open file %ssav.pp%s for reading\n",
          YELLOW, NORMAL);
        continue;
      }
      while (fgets(line, sizeof(line), loadfile)) {
        printf("%s", line);
        if ((ii+1) % 14 == 0) {
          pagePrompt();
          key = getc(stdin);
          if (key == 'q') { break; }
        }
      }
      fclose(loadfile);
    }
    else if (testCom == SAVELAST) {
      FILE * savefile;
      if ((savefile = fopen ("last.pp", "w")) == NULL) {
        printf ("Could not open file %slast.pp%s for writing \n",
          YELLOW, NORMAL);
        continue;
      }
      saveAssembledProgram(&m.program, savefile);
      fclose(savefile);
    }
    else if (testCom == CHECKPROGRAM) {
      validateProgram(&m.program);
    }
    else if (testCom == CLEARPROGRAM) {
      clearProgram(&m.program);
    }
    else if (testCom == CLEARLAST) {
      // need to set last instruction to undefined
      newInstruction(&m.program.listing[m.program.count-1], UNDEFINED);
      m.program.count--;
      printSomeProgram(&m.program, 7);
    }
    else if (testCom == INSERTINSTRUCTION) {
      // insert an instruction at the current program ip
      // need to recalculate jump address (add 1)
      printf ("Inserting instruction at %s%d%s \n",
        YELLOW, m.program.ip, NORMAL);
      insertInstruction(&m.program);
      printProgram(&m.program);
    }
    else if (testCom == EXECUTEINSTRUCTION) {
      execute(&m, &m.program.listing[m.program.ip]);
      showMachineTapeProgram(&m, tapeContext);
      /*
      printSomeProgram(&m.program, 6);
      printf("%s--------- Machine State ----------%s \n",
         YELLOW, NORMAL);
      showMachine(&m);
      */
    }
    else if (testCom == PARSEINSTRUCTION) {
      struct Instruction * ii;
      struct Label table[100];  // void jumptable
      printf("testing instructionFromText()...\n");
      if (strlen(args) == 0) {
        printf ("%s No example instruction given... %s \n"
                " Try %spi: add {this}%s \n", GREEN, NORMAL, YELLOW, NORMAL);
        continue;
      }
      newInstruction(ii, UNDEFINED);
      instructionFromText(stdout, ii, args, -1, table);
      printEscapedInstruction(ii);
      printf("\n");
    }
    else if (testCom == TESTWRITEINSTRUCTION) {
      printf("Testing writeInstruction()... \n");
      writeInstruction(&m.program.listing[m.program.ip], stdout);
    }
    else if (testCom == STEPCODE) {
      printf("step through next compiled instruction:\n");
      step(&m);
      printSomeProgram(&m.program, 6);
    }
    else if (testCom == RUNCODE) {
      printf("%sRunning program from instruction %s%d%s...%s\n",
        GREEN, CYAN, m.program.ip, GREEN, NORMAL);
      run(&m);
      printf("\n%s--------- Machine State ----------%s \n",
         YELLOW, NORMAL);
      showMachine(&m, TRUE);
    }
    else if (testCom == RUNZERO) {
      printf("%sRunning program from ip 0%s\n", GREEN, NORMAL);
      m.program.ip = 0;
      run(&m);
    }
    else if (testCom == RUNTOTRUE) {
      printf("run program until the flag is set to true\n");
      runUntilTrue(&m);
      showMachineTapeProgram(&m, 2);
    }
    else if (testCom == RUNTOWORK) {
      if (strlen(args) == 0) {
        printf ("%sNo text given to compare%s \n", GREEN, NORMAL);
        printf ("%sUsage: rrw %s<text>%s \n", GREEN, YELLOW, NORMAL);
        printf ("   runs the current program until workspace is <text> \n");
        continue;
      }
      printf("Running program until workspace is \"%s%s%s\" \n",
         YELLOW, args, NORMAL);
      runUntilWorkspaceIs(&m, args);
      // printf("%s--------- Machine State ----------%s \n",
      //   YELLOW, NORMAL);
      showMachineWithTape(&m);
    }
    else if (testCom == RUNTOENDSWITH) {
      if (strlen(args) == 0) {
        printf ("%sNo text given to compare%s \n", GREEN, NORMAL);
        printf ("%sUsage: rrw %s<text>%s \n", GREEN, YELLOW, NORMAL);
        printf ("   runs the program until workspace ends with <text> \n");
        continue;
      }
      printf("Running program until workspace ends with \"%s%s%s\" \n",
         YELLOW, args, NORMAL);
      runUntilWorkspaceEndsWith(&m, args);
      showMachineTapeProgram(&m, tapeContext);
    }
    else if (testCom == IPZERO) {
      m.program.ip = 0;
      printSomeProgram(&m.program, 4);
    }
    else if (testCom == IPEND) {
      m.program.ip = m.program.count-1;
      printSomeProgram(&m.program, 4);
    }
    else if (testCom == IPGO) {
      if (strlen(args) == 0) {
        printf ("%s No number given to jump to %s \n", GREEN, NORMAL);
        continue;
      }
      int ipnumber = 0;
      int res = sscanf(args, " %d", &ipnumber);
      if (res < 1) {
        printf ("%s Couldnt parse number %s \n", GREEN, NORMAL);
        continue;
      }
      m.program.ip = ipnumber;
      printSomeProgram(&m.program, 4);
    }
    else if (testCom == IPPLUS) {
      m.program.ip++;
      printSomeProgram(&m.program, 4);
    }
    else if (testCom == IPMINUS) {
      m.program.ip--;
      printSomeProgram(&m.program, 4);
    }
    else if (testCom == SHOWSTREAM) {
      // show some upcoming chars from stream and then
      // reset stream position
      long pos = ftell(m.inputstream);
      int num = 40;
      char c;
      int ii = 0;
      printf ("%sNext %s%d%s chars in input stream...%s\n",
        GREEN, YELLOW, num, GREEN, BLUE);
      while (((c = fgetc(m.inputstream)) != EOF) && (ii < num)) {
        printf("%c", c);
        ii++;
      }
      printf("%s\n", NORMAL);
      fseek(m.inputstream, pos, SEEK_SET);
    }
    // deal with pure machine commands (not testing commands)
    else if (textToCommand(command) != UNDEFINED) {
      void * p = NULL; // no jump table here
      if (mode == IPCOMPILE) {
        compile(&m.program, line, m.program.ip, p);
      }
      else if (mode == INTERPRET) {
        // execute the entered command but dont insert it
        // in the program.
        struct Instruction ii;
        instructionFromText(stdout, &ii, line, 0, p);
        execute(&m, &ii);
        // execute automatically advances ip pointer, which we
        // dont want in this case. Jumps will probably not be entered
        // interactively so, should not cause trouble here.
        m.program.ip--;
        showMachineTapeProgram(&m, tapeContext);
        continue;
      }
      else {
        compile(&m.program, line, m.program.count, p);
      }
      // check if the instruction is ok
      // checkInstruction(&m.program.listing[m.program.count-1], stdout);

      if (mode == COMPILE) {
        printSomeProgram(&m.program, 4);
      } else step(&m);

      m.program.ip = m.program.count;
      printSomeProgram(&m.program, 5);
    }
    else if (testCom == EXIT) {
      printf("%sSaving program to '%slast.pp%s'...\n",
        GREEN, YELLOW, GREEN);
      FILE * savefile;
      if ((savefile = fopen ("last.pp", "w")) == NULL) {
        printf ("Could not open file %slast.pp%s for writing \n",
          YELLOW, NORMAL);
        continue;
      }
      saveAssembledProgram(&m.program, savefile);

      fclose(savefile);
      fclose(inputStream);
      colourPrint("Freeing Machine memory...\n");
      freeMachine(&m);
      colourPrint("Goodbye !!\n");
      exit(1);
    }
    else {
      printf("%sUnrecognised command:%s %s %s \n",
        BROWN, command, GREEN, NORMAL);
    }

  }

  //newTape(&t); printTape(&t);
  fclose(inputStream);
  return(0);
}
// </code>

/*

  OTHER PARSING IDEAS

    The code below implements a shift-reduce parser and a token
    lexer in a reasonably simple format.

    Another idea for parsing: a token/attribute structure
    and an open stack to hold it.
    Then compare token sequences

    lexing phase using fns such as scanargument, scancommand ...
    for some tokens, standard c scanf() fn is ok
    parsing phase uses a reduction loop to implement bnf shift-reduce
    grammar rules...

      // use a pointer to last item on stack eg
      // item * last = tokens[end];
      struct item { char * attribute; enum Token token; }
      enum Token = {ARGUMENT, LBRACE, RBRACE, COMMAND ... };
      // tokenInfo array with information about each token type
      // scanArgument etc works just like scanParameter in the machine code
      // read loop:
      while (more chars in input stream) {
        // ----------------
        // lexing phase. Lexical analysis.
        // could use a switch on nextChar
        if (nextchar == '{') {
          // no attributes (values for {
          itemarray[end]->token = LBRACE;
          // end++
          last++  // increment pointer
        }
        if (nextchar == '}') {
          // no attributes (values for {
          // last->token = RBRACE;
          itemarray[end]->token = RBRACE;
          last++
        }
        if (nextchar == '[') {
          itemarray[end]->token = ARGUMENT;
          // nextChar = scanArgument(itemarray[end])
          itemarray[end]->ttribute = scanArgument(..);
          end++
        }
        if (nextchar == [a-z]) {
          itemarray[end]->token = COMMAND;
          itemarray[end]->ttribute = scanArgument(..);
          end++
        }
        if (nextchar == [:space:]) { scanSpace(..); }

        // ---------
        // parsing phase
        // the token reduction loop (corresponds to bnf grammar
        // reduction.
        // The reduction loop needs to keep going until no more
        // reductions are found.
        int reduction = 0;
        while (reduction) {
           reduction = 1;
           // or fn: stackend(COMMAND, ARGUMENT)
           // or if (stackend(&tokstack, COMMAND, ARGUMENT))
           // or if (last[-1]->token == COMMAND && last->token == ARGUMENT) {
           if (tokenarray[end-1].token == COMMAND &&
               tokenarray[end].token == ARGUMENT) {

              // get items off stack, manipulate attributes then
              // put new tokenitem on stack with new attribute
             reduction = 0;   // check all other reductions
             tokenarray[end-1].token == INSTRUCTION;
             //...
             // some string manipulation on attributes
             // do malloc on attrib if not enough string space
             strcat(tokstack[end-1].attrib, tokstack[end].attrib);
             end--;
           }
           // ... more reductions
         } // while more reductions

      } // read input stream loop


    This technique could be used to parse the script file

  ----------------

*/
```