CSI 333 – Programming at the Hardware/Software Interface – Fall 1999
Programming Project 1

WARNING: Read and follow the instructions herein about Revision Logs before you start this project. Otherwise, you might get ZERO credit or face more serious consequences.

1 Objectives:

(1) Practice programming control operations and manipulating character array and other data in RISC (MIPS) assembly language. (2) Understand recursive procedure and stack frame (activation record) implementation, operation and usage. (3) Begin learning about regular expression pattern matching with special cases. (4) Use a given software design outline, an incremental implementation/test/debug plan and a version archive in order to learn to apply elementary software engineering methods yourself to future projects.

2 Advice

This project is designed to be done after you did recursive procedure experiments from Lab Exercises 2. Therefore, if you didn’t have your lab session yet, you should work on the Lab Exercises up to checkoff 3 (or beyond) and then begin this project. Also, review lecture slides and textbook material on byte (character) array processing, the ASCII table (since you will have to express numeric character constants in hex or decimal), and procedure linkage including the use of the stack frame to save and restore the register contents that require saving.

Before you attempt to write even one line of code, you must read and analyze the problem statement (“specification”) to understand what your program is supposed to do, and the design and implementation plans to understand how to go about creating the program. Read the whole assignment and submit any questions to the lecture, the newsgroup (sunya.classes.csi333), a TA or the professor.

The numbered items outline a design for the modules and a plan for adding new functionality to partially implemented modules. Non-trivial programming work projects are best done incrementally. This means first write, test, debug and test again (after every change, no matter how little it is) a version that meets just a few of the specifications correctly. The first and subsequent versions enable you to add new functionality a little at a time, in such a way that each time new code is added, you can trust the previously completed code to help test and debug the new code.

Copies of all tested previous versions should be kept as files whose names are given after the version description. They of course have NO SYNTAX ERRORS (the assembler or compiler accepts and successfully runs or compiles them) because they had to have been assembled or compiled for testing! This way, if you mess up the newer version beyond repair, you can start again with a previous version.
Revision Log

You must maintain a “revision log”. At the beginning of the file, after your name, student ID number and TA name, write a comment for each time you (1) began writing a new version, (2) fixed a bug, and (3) concluded by testing that the current version is correct; the comment must specify the version number, date and approximate time.

Start a new version file by making a copy of the the previous version file with a new name. You will submit the last version you complete, which then contain the revision log for all the versions you did.

A submission without a revision log will receive ZERO credit, and logs found to be faked will be treated as evidence of CHEATING.

Specification Summary

Pattern matchers are useful for testing a string for particular properties, such as whether it begins with a particular letter or has a particular substring in it. Pattern matching can then called repeatedly to find from a set of strings all the strings we might be interested in. To see pattern matching in action, go to /usr/bin (with cd /usr/bin) and try the command “Is *es*” Then, try “Is” and see how hard it is find all the Unix commands whose names contains the consecutive letters es. A real, standard Unix utility pattern matcher is described by man regexp. Pattern matching is a heavily used feature of the programming language Perl which is commonly applied to interpret what people type into response boxes on Web pages. Regular expression patterns are also used in programming language manuals or other application specifications to describe precisely what the legal strings for various applications are.

The project program will prompt\(^1\) for and accept a “pattern string” followed by a “subject string.” Each string comes from standard input. If any input exceeds 40 characters not including the newline character that the user types to end it, the program should not fail (crash, loop or recurse infinitely, erase your files, etc.) but should print a message and exit.

The pattern matcher will test whether the entire subject string has the properties defined by the pattern. The subject and pattern match when the subject can be broken into substrings, one substring for each pattern character, in such a way that each substring obeys the rule below for the corresponding pattern character.

The subject (string) can contain any characters. The pattern can contain only upper or lower case letters, the digits 1 or 2, or the “unlimited wildcard” character *. When an illegal pattern character is detected a message is printed and the pattern will not match any subject. Here are all the rules:

1. An upper case letter pattern character matches only one substring: The single character equal to the pattern character. For example, pattern ABCA matches only one string:ABCA Here, the subject is broken into substrings A, B, C and A.

2. The pattern character * matches any substring at all, including the empty string. For example, *ABC* matches ABCA

   and Join the ABCA (since the first * matches Join the and the second matches the empty

\(^1\)Details like the particular prompt strings will given in the implementation plan rather than clutter the specification summary
string between C and A. It also matches Learn your ABCs at SUNY A but not ABBCA, ABCABC or A.

Pattern *ES* matches TEST, YES, UNCOMPRESS and I'm the BEST.

3. A lower case letter pattern character matches a length one substring that consists of the pattern character or its upper case version. For example, pattern funnY matches funnY, FUNnY, fUNnY, FUNnY and nothing else.

4. The pattern character 1 (the digit) matches any one subject character (i.e. substring of length 1). For example, pattern 1And matches AND, SAnd, rAnd, $And and Band; but not And or zzAnd.

5. The pattern character 2 (the digit) matches any substring of length exactly 2. For example, pattern a2A1 matches subjects A$Aq, abcAZ, A12A1, etc., but not aAaA, aAaaA (since pattern character A doesn't match the 4th subject character a), A2A1, etc.

If the pattern matches the subject print Yes, it matches
Otherwise, print No, it doesn't match.

Since one objective is to learn about procedure calling and recursion, reports of call and return operation described below must be printed to display the algorithm in operation.

Design Outline

The procedure at _start will print the prompts, accept the pattern and subject strings, call the procedure with C-like declaration

```c
int match( char *ppattern, char *psubject );
```

which returns 1 if the subject matches the pattern and 0 otherwise. The two strings will be stored in arrays of 42 characters. Each string, if legally inputted, will have a single ASCII newline (decimal 10) followed by a null to terminate it. The procedure then prints the result message and exits. _start might call one procedure twice to read the input strings; this procedure will print a message and exit if an input is too long.

The match procedure implements the recursive algorithm outlined in the Implementation Plan. Therefore, match will use the system stack to save return addresses, and register contents that require saving. match will use procedures to test for various cases of characters, convert an upper case letter to its lower case version, and possibly other minor operations of your choice.

Implementation Plan

Version 1:

Input

Print the prompt (and a newline)
Type pattern and subject on separate lines:

Accept two lines of input. Use the (x)spin read_string syscall to read each string. Read in Stroustrup, your other C++ books, or online documentation about the standard C fgets function
because `read_string` works like `fgets`. Design two static storage areas for the input strings to accomodate inputs of 0 to 40 characters not including the newline and null character.

If 41 or more characters are typed on one line, the program should immediately print `Input too long.` and quit (exit syscall). Programming tips: Request a maximum transfer of 42 characters (including the null) by putting 42 in `$a1`. Use a buffer at least 42 bytes long of course; don’t bother to clear or initialize it. Then, program a loop to scan the buffer for the first character that is either a newline (decimal 10 in Unix) or a null (value 0). If the input transferred from the user is terminated by null rather than a newline, then the user typed too much before pressing the enter key.

The pattern matcher will rely on the newline character at the ends of the pattern and the subject strings.

Waldron’s `length.a` example illustrates the kind of loop you will need. Don’t just copy it: Your program must solve a different problem. It would be a good idea to implement one procedure that reads an input string into memory at a parametrized address and checks the length (to be called twice in this application). But, don’t let this hold you up if you are not yet fluent enough with assembly language procedures to do it quickly.

If the two input strings are not too long, the code should call a procedure labelled `match` with 2 parameters: the addresses of the first bytes of the pattern and subject strings respectively.

**Match Result Output**

`match` will be developed to return (in the register for procedure return values under the MIPS procedure linkage conventions) integer 1 if the subject matches the pattern and 0 otherwise. Start by writing `match` to do something very simple, like test whether the first characters of the pattern and subject are equal, or even simply return a constant, just to test the result printing code. Dummy procedure implementations are sometimes called “stubs”.

If the subject matches the pattern, print (followed by a newline)

Yes, it matches.
Otherwise, print (again with a newline)

No, it doesn’t match.

Then make the program exit.

Save your tested version with file name `matcher1.s`

**Version 2: Procedure Linkage Management**

**Debugging Flag**

Create a “debugging flag” memory variable with

```
.globl DEBUG
```

```
DEBUG: .word 1 #1 for debugging, 0 for production
```

**Documentation of Stack Frame Layout**

The piece of code at the beginning of a procedure that allocates the stack frame, saves registers, sometimes copies arguments into the stack and may do other linkage tasks is called the `prologue`.  

4
The code before the return operation that pops the stack, etc., is called the procedure’s *epilogue.*

Begin a section of comments under the label `match:` which documents how `match` will use its stack frame: Decide at which displacement from `$sp$ (0($sp$), 4($sp$), 8($sp$), etc) each register will be saved, for the registers that need saving.

Write the first version of `match`'s prologue and epilogue. KEEP the prologue, epilogue and their documentation CONSISTANT in all later versions. As you add more and more code, you may have to save more and more registers. You might chose to also implement local (automatic) procedure variables in the stack frame. Document them too.

**Procedure Activation/Return Trace Output**

Write code immediately after the prologue that prints:

```
matcher ACTIVATED ($sp=...) for pattern:pattern
   subject:subject
```

and a blank line. Here, “...” denotes the decimal numeral that is obtained when `print_integer` is applied to the value in register `$sp$`. “Documentation syntactic variables” `pattern` and `subject` are placeholders for the particular pattern and subject strings for which procedure `match` was called. Adjust² the number of spaces in the second line so that the two colons line up. It is much easier to tell if the pattern and subject match if the colons line up.

Write code immediately before the epilogue that prints:

```
matcher ($sp=...) RETURNING for pattern:pattern
   subject:subject
   return value:$N$
```

and a blank line. Print the value in `$sp$` and adjust the spaces. `pattern` and `subject` are the same strings that had been printed by the ACTIVATION report of the SAME activation that is returning now. Tip: Save the argument registers in the stack frame. $N$ denotes the value (0 or 1) that will be returned (after the epilogue finishes).

The activation and return report code should be executed conditionally on the value in memory location `DEBUG` being non-zero.

Save the tested version (with `DEBUG`: containing 1) in `matcher2.s`

**Version 3: Literal Pattern Characters**

Now implement the first case of pattern character types: An upper case letter pattern character matches a single subject character equal to itself. For example, pattern `ABCD` matches only `ABCD`. Subjects like `ABCD`, `AB`, `ABABCD`, etc., do NOT match with pattern `ABCD`.

The pattern matching algorithm for this project is recursive. Here are the cases needed so far:

- If the pattern string is just the newline character it means the pattern is “empty”. The only subject matched by the empty pattern is the empty string. There will be only one newline character in the pattern and subject strings, since the subject is terminated by newline too³. Therefore, when the matcher detects the newline at the beginning of the pattern, it should

---

²Do **HAND** trial and error or calculation after you observe the length of the decimal numeral: Do NOT program this!!

³There should also be the null character terminator right after the newline so that the `print_string` syscall will print the strings properly.
check whether the first character $F$ of the subject is the newline. Return 1 if $F$ equals newline and 0 otherwise.

- If the pattern string begins with a character that is specified to match subject substrings of fixed length, check if the appropriate number of subject characters at the beginning of the subject string do match according to the specification. Return 0 if not. Otherwise, call the matcher recursively to tell whether the rest of the pattern matches the rest of the subject. Use the value returned by the recursion to determine the correct value to return.

The first two algorithm cases cover our simplest pattern character type. If the first pattern character is an upper case letter, the “appropriate number” is one, and the algorithm should test if the first subject character equal that upper case pattern character. To get the address of the rest of the pattern or subject string, just add 1 to the address of the first character.

Illegal Pattern Characters

When procedure `match` detects a character in the pattern whose meaning wasn’t specified or implemented yet, `match` should print

**Illegal pattern character.**

and return with the value 0 (which indicates the match was unsuccessful).

Save as `matcher3.s`

Version 4: Bounded Length Wildcards

Add two more cases: The digit character 1 pattern character matches any one (non-newline) subject character. The digit character 2 matches any two adjacent (non-newline) subject characters.

Save as `matcher4.s`

Version 5: Unbounded Wildcard Pattern Character

Add another case: The asterisk * pattern character matches any number (0, 1, 2 or more) of any adjacent (non-newline) subject characters.

For example, pattern `MATCHER*A` matches `MATCHER1.A`, `MATCHER2.A`, `MATCHER29.A`, `MATCHER`, `MATCHER1999end()`A, etc.

This requires another algorithm case:

- Suppose the first character of the pattern is specified to match strings of various lengths. Follow the pseudocode:

```c
for ( i = 0; i <= length of subject; i++ )
{ if( the length i prefix of subject is matched by the first character of the pattern)
  {
    apply `match` recursively to the rest of the pattern and the rest of the subject;
    if the recursion returns success, return success;
  }
}
return failure; (since the loop found no way to match)
```

Notice that the $i=0$ in the first iteration. This means the length $i$ prefix of the subject is the empty string, so the “rest of the subject” is the entire subject string. However, the “rest
of the pattern" supplied to the recursive calls to match is always the suffix that follows the
first pattern character. Hence the lengths of the pattern strings passed to match continually
decrease as the recursion gets deeper and deeper.

Remark: There might be several different ways the pattern can match the subject. For example,
pattern *A*Z matches AAAZ three ways: The first * may match the empty string, A or AA. The A
in the pattern will match the 1st, 2nd or 3rd A of the subject string respectively. A pattern that
can match some string more than one way is called ambiguous. Our pattern matcher is required
to just find one of the ways no matter how many other ways exist.

Save as matcher5.s

**Version 6: Character Class Literals**

Add another case of pattern character: A lower case letter in the pattern matches itself or its upper
case version in the subject. This means a lower case letter type character in the pattern matches
a subject substring whose length is exactly 1.

The logic is very similar to what is in version 3. You should create and use procedures to (1)
check if a character is an lower case letter, (2) to check if a character is an upper case letter, and
(3) convert any letter into its lower case version if it is not lower case already (you must figure out
what constant to add).

Save as matcher6.s

**Test Cases, Grading etc.**

1. By 12:01 AM Wednesday, October 6, a directory of sample inputs and correct outputs for indicated
versions will be published in the ECL class account at pathname ~csi333/Project1/Samples
This material will also be put on the class web site.

2. The due time is Wednesday, October 20, 9:00AM. One file of MAL assembly code named
either matcher1.s, matcher2.s, ..., matcher6.s is to be submitted electronically. It MUST
contain the revision history comments for ALL the work, otherwise you may get a 0 for the
entire project. Correct test case output will not count if it is inconsistent with the revision
history comments.

3. Late submissions will be accepted until Monday, October 25, 9:00PM (5,500 days). However,
a lateness penalty factor equal to \((10.0 - ND)/10.0\) where \(ND\) is the amount of time late,
measured in days, will be multiplied into your score, computed using floating point arithmetic.
This means the amount deducted for lateness will begin at 0% and rise (almost) continuously
to 55% during the late turnin period. Early submissions will be accepted but will not earn
any bonus.

4. 70% of the score will come from evaluation of outputs on test cases. To get any of these
points, your MAL file must be accepted without error messages by the spin simulator. Give
it a final test before you submit it; do NOT open it with an editor AT ALL after the final
test.

5. 30% of the score will come from internal documentation:
• Identification: Your full name, your TAs name, your section number.
• Quality (clarity, accuracy, completeness, etc.) of revision history.
• Consistent indentation.
• Procedures/functions: What each does in terms of parameter register contents, return value, and action on any other data it uses.
• Layout of each procedure’s stack frame.
• Usage of registers or memory variables expressed as invariants in loops or conditional code blocks.

6. Note the section number of the lab you are registered in. When you are ready to submit your work, go (use cd) to the directory containing the matcher*.s file you plan to submit. Type the command
   
   turnin-csi333 -p Matcher -c csi333 filename
   
   where filename is matcher6.s, matcher5.s, etc. (whichever version you completed). (Use /opt/bin/turnin-csi333 if the command wasn’t found.) After you issue this command, the system responds with

   The sections of csi333 are:

   sec1461
   sec1462
   ...

   Enter your section:

Now type the name of the section you are registered in. The system responds with

   Your files have been submitted to csi333, Matcher for grading.

   Please note that using the turnin program as above is the ONLY acceptable way of submitting programming assignments in this course. You should NOT mail the files to the instructor or the TAs.

7. Pay attention to all error messages printed by turnin: An error message probably means your submission will not be received. Recheck the instructions, repeat until turnin use with
   
   --v described below shows successful receipt, and see a TA if positive acknowledgment is not obtained.

8. If you use the turnin command above again at a later time, then the files submitted previously would be replaced by the newly submitted files. (This allows you to resubmit a program if the previous submission was erroneous. But it wipes out the submission time record of the previous submission, so your revised submission will be more late.)

9. Strongly Recommended!! After any submission, run
   
   /opt/bin/turnin-csi333 -c csi333 --v
   
   and observe the report of what files were received. If the file(s) you meant to send are not listed, check and redo the instructions. See a TA if you cannot get a positive acknowledgment.