CSI 333 – Programming at the Hardware/Software Interface – Fall 2000
Programming Project 3

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. (5) Begin to follow formal testing requirements.

2 Advice

This project is designed to be done after you did the system stack frame maintenance and recursive list traversing exercise from Lab Exercises 2. Also, review lecture slides and textbook material on byte (character) array processing and procedure linkage including the use of the stack frame to save and restore the register contents that require saving. You will not need the ASCII table (since you can express numeric character constants with the ‘.’ syntax in spim as well as C/C++.

Before you attempt to write even one line of code, you must read and 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), any TA (not just your section leader) 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.

3 RCS and Revision Logging

The instructions and requirements about use of RCS and revision logging from programming project 2 must be followed for this project. The assembly language file name for this project should be “matcher.a”. The project name to be used with the -p option of the turnin-csi333 and turnin-csi333 -v commands is “matcher”. We will expect to see your revision history log embedded in your matcher.a file as comments inserted by RCS.

Copies of all tested previous versions will be kept in the RCS database. Each version that passed the tests for that version will 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.

4 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 be invoked 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 “ls *.es*” Then, try “ls” and see how hard it is find all the Unix commands whose names contains the consecutive letters es.

Try the commands “ls [ab]*”, and “ls [ab]*e”. What names do these patterns select? Why?

The Unix utility pattern matcher is described by man regexp. Pattern matching is a heavily used feature of the programming language Perl which is commonly used in “CGI programs” 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. A language (set of strings) that can be defined as the set of all strings that match a particular regular expression is called a regular language. The fact that regular languages can be defined just as well by scanners with a finite number of states (“finite state automata”) is fundamental to the theory of computation.

The project program will prompt¹ 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 expression, in such a way that each substring obeys the rule below for the corresponding pattern expression.

The subject (string) and pattern strings can contain any characters except for embedded newlines. The newline character will be reserved to mark the end of the subject and pattern strings. In the pattern string, all characters are treated alike except for the following 5 special characters: [ ], { }, and \ . All other characters are called “ordinary characters”.

Here are all the rules:

1. Each ordinary character (except within { } or [ ] ) in the pattern is a pattern expression that matches a single copy of itself in the subject string.

   Therefore, a pattern of ordinary characters only matches only itself.

2. A substring in the pattern beginning with [ and ending with ] and with ordinary characters in between is an expression that matches exactly one subject string character equal to one of the ordinary characters between those [ and ]. For example, [abc] matches “a”, “b” and “c”.

¹Details like the particular prompt strings will given in the implemnet plan rather than clutter the specification summary.
3. A substring in the pattern beginning with {, ending with } and with ordinary characters in between is an expression that matches any number of consecutive subject string characters, including none, each of which is any one of the ordinary characters between those { and }. For example, \{abc\} matches the empty string, “a”, “baa”, “bac”, “abc”, “abccccc”, etc.

4. A pattern formed by \(n\) expressions concatenated together matches strings formed by concatenating \(n\) strings, for which the \(i\)th string matches the \(i\)th expression for \(i = 1, 2, \ldots, n\). For example, pattern “\{C{+} is, {abc}[?;:!]\}” matches “\(C\) is, aaaaaabbbbbbbbbaaaaaaa!”, “\(C++\) is, ba?”, “\(C++\) is, ;;”, etc. In the third example, \{abc\} matched the empty string between the “,” and the “;”.

5. The special character \(\backslash\) is called the “escape character.” It enables the pattern writer to use a special character anywhere an ordinary character would be acceptable. Thus, outside of expressions [...] or {...}, the five length 2 character sequences \(\backslash\backslash\), \(\backslash\), \(\backslash\) \(\backslash\) \(\backslash\) \{ and \} respectively. These five length 2 sequences are called “escape sequences”.

    The appearance of (an unescaped) \(\backslash\) in the pattern immediately followed by the end of the pattern or a character that is not \(\backslash\), [ ] \{ or \} makes the pattern illegal.

6. Escape sequences can appear within [...] or {...} expressions. In that context, the escaped character is treated as an ordinary character within the set that the [...] or {...} denotes.

7. Unescaped special characters cannot appear within the ... of the [...] or {...} expressions, and expressions beginning with [ or { must be complete. Thus, “[a{bc}d]”, and “[a]” and “{abcd}” are illegal patterns.

    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 operations described below must be printed to display the operation of a recursive algorithm.

**Design Outline**

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

\[
\text{int \ match( char *ppattern, char *psubject );}
\]

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 input procedure twice to read the input strings; the input 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 and possibly other minor operations of your choice.
5 Implementation Plan

5.1 Version 0: Input, Output and Procedure Linkage Management

5.1.1 Input

Print the prompt (and a newline)
333 matcher Version n 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
H/S, 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.

5.1.2 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.

5.1.3 Debugging Flag

Create a “debugging flag” memory variable with
.globl DEBUG
5.1.4 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. 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.

5.1.5 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 $sp$ after $sp$ is decremented to allocate this activation's stack frame. “Documentation syntactic variables” pattern and subject are placeholders for the particular pattern and subject strings for which procedure match was called. Adjust\(^2\) 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 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.

5.2 Version 1: Literal Pattern Characters

Now implement the first case of pattern character types: An ordinary 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:

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

5
• 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 too\(^3\). Therefore, when the matcher detects the newline at the beginning of the pattern, it should check whether the first character \(F\) of the subject is the newline. Return 1 if \(F\) equals newline and 0 otherwise.

Test cases \(t11\) and \(t12\): Empty pattern, empty subject (should match). Empty pattern, non-empty subject (should not match).

• If the pattern string begins with a character that is specified to match subject characters, 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 \textit{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.

Test cases: \(t1\) * Combinations of non-empty patterns of length 1, 2 and more with matching subjects, empty subjects, subjects that fail because of the first character, subjects whose first character matches but other characters don’t, cases when the pattern or subject are prefixes of another.

The first two algorithm cases cover our simplest pattern character type. If the first pattern character is an ordinary character, the “appropriate number” is one, and the algorithm should test if the first subject character is equal to first 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.

### 5.2.1 Illegal Pattern Characters

When procedure \texttt{match} detects a character in the pattern whose meaning wasn’t implemented yet, or is otherwise illegal, \texttt{match} should print

\texttt{Illegal pattern character},

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

More \(t1\) * cases for patterns with illegal characters.

### 5.3 Version 2: Pattern Subset to Match One Subject Character.

Similar to the algorithm case of version 1, except the matcher must compare the first subject character to each of the characters in the [...] sequence until it finds an equality match or reaches the ] of the [...] sequence.

Test cases named \(t2\) *: Empty [] in pattern matches nothing. [...] at the beginning, middle, end; [...] with one and with more characters, multiple [...]’s combined with matching and non-matching subjects.

Remark: The searches for these versions do not do any \textit{backtracking}. The search proceeds left-to-right along the subject string. Each subject string character is matched against the each expression (literal or [...] expression) in the pattern. If that character match fails, the whole string match fails. If that character match succeeds, the success or failure of the whole string match is just the success or failure of the match of rest of the subject and the rest of the pattern.

\(^3\)There should also be the null character terminator right after the newline so that the \texttt{print,string} syscall will print the strings properly
A \{ \ldots \} expression of Version 3 can match many different substrings at the beginning of a subject string. For example, consider pattern \{AB\}A\{BC\} and subject BBACB. The expression \{AB\} can match the empty string, B or BB or BBA. When the matcher tries to find a match where \{AB\} matches the empty string, it will be successful in matching the two B’s in BBACB with \{BC\}, but will discover failure when it processes the A in BBACB. That’s because A is not matched by \{BC\}.

Therefore, the choice to match \{AB\} with the empty string is wrong. Similarly, matching \{AB\} with BBA is also wrong since the A in BBACB must match the literal A in \{AB\}A\{BC\}.

The version 3 algorithm will do backtracking.

5.4 Version 3: Pattern Subset to Match Zero or More Subject Characters

The matcher procedure activation should detect that the first pattern character (of the pattern string this activation was called on) is \{.

- Find the } that closes the subset beginning with the \{. Calculate the address of the first pattern character beyond this \{. This is the pattern address to use in one of the recursive calls to matcher that will be made by this activation.

- (Try to complete the match with the \{\ldots \} set matching 0 characters). Call matcher recursively with given subject string and the pattern substring that begins just after the \{\ldots \} set. If the recursive match is successful, return success. Otherwise, go on to the next step.

- Check if the subject string is non-empty and the first subject string character is in the subset written at the beginning of the pattern.

  If so, call matcher recursively on the rest of the subject string and the given pattern string.

  (The recursion will handle the case of more subject characters matching the subset being processed here!) Return the result of the recursive call.

  If not, return 0 for match failure.

Notice that for each recursive call, one or both of the lengths of the pattern string and the length of the subject string was reduced.

Hence the sum of the lengths of the strings passed to match continually decrease as the recursion gets deeper and deeper, and the recursion terminates.

Remark: There might be several different ways the pattern can match the subject. For example, pattern \{A\}A\{A\}Z matches AAAZ three ways: The first \{A\} 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.

5.4.1 Test Cases

Test cases for Version 3 should have names of the form t3\*. They should include all combinations of unsuccessful and successful matches cases that require backtracking or not, plus some cases of successful and unsuccessful matches against ambiguous patterns.

5.5 Version 4: Escape sequences outside of sets

Submit appropriate test cases named t4\*. 7
5.6 Version 5: Escape sequences outside or inside of sets

Submit appropriate test cases named t5*.

6 Test Cases, Grading etc.

1. A directory of sample inputs correct outputs, and a testing script for indicated versions will be published in the ECL class account at pathname~csi333/Project3/Samples Watch for the announcement on the class Web page and newsgroup.

2. The due time is Wednesday, November 1, 9:00PM. The RCS database named matcher.a,v for one file of MAL assembly code name matcher.a, plus your test case files and all scripts needed for running your tests 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, November 6, 9:00PM (5,000 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 50% during the late turnin period. Early submissions will be accepted and will earn a 5% bonus.

4. 55% 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 spim simulator. Give it a final test before you submit it; do NOT open it with an editor AT ALL after the final test.

5. 20% of the score will come from the test cases that you submit, what cases they cover and whether the scripts you submit automate the testing.

6. 25% of the score will come from internal documentation:

   • 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.

$Id: pr3.tex,v 1.3 2000/10/17 13:20:29 sdc Exp sdc $