First Midterm Exam
CS164, Fall 2009
Oct 20, 2009

- Please read all instructions (including these) carefully.
- Write your name, login, and SID.
- No electronic devices are allowed, including cell phones used as watches.
- Silence your cell phones and place them in your bag.
- The exam is closed book, but you may refer to one (1) page of handwritten notes.
- Solutions will be graded on correctness and ***clarity***. Each problem has a relatively simple and straightforward solution. Partial solutions will be graded for partial credit.
- There are 16 pages in this exam and 3 questions, each with multiple parts. If you get stuck on a question move on and come back to it later.
- You have 1 hour and 20 minutes to work on the exam.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. **Do not** use any additional scratch paper.

LOGIN: _____

NAME: _____

SID: _____

Problem	**Max points**	**Points**
1	**40**	
2	**25**	
3	**35**	
TOTAL	**100**	

Problem 1: Extend regular expressions with subtraction [40 points]

In the question you will extend regular expressions with an operator that subtracts languages. For now, let's call this operator *sub*. A string s matches regular expression R1 sub R2 if it matches R1 and does NOT match R2. That is, s matches R1 sub R2 if and only if s is in the set of strings L(R1) \ L(R2), where \ is the set subtraction operator, and L(R) is the language (set of strings) matched by the regular expression R.

Motivation. One motivation for this operator is to describe the set of identifiers in a programming language, which does not include a special set of keywords like if, for, etc. For example, consider the strings accepted by our calculator language from HW2. One of the productions for that grammar was Id ::= [_a-zA-Z][_a-zA-Z0-9]* but not the words 'SI' or 'in'. Our current regular expression operators give us no good way to express this. But with this this new operator we could write

```
Id ::= [_a-zA-Z][_a-zA-Z0-9]* sub (SI|in)
```

Part 1: Syntax for the subtraction operator [6 points]

Your goal is to come up with suitable syntax for the sub operator. Why not use *sub* as a keyword? For the same reason why we use |, rather than *or,* to denote the alternation operator.

Extend the grammar below with your operator.

```
R -> CHAR
 | R '*'          %dprec 1
 | R R            %dprec 2
 | R '|' R        %dprec 3
 | '(' R ')'

 ;

CHAR -> /[A-Za-z0-9]/ ;
```

Part 2: Precedence [7 points]

You must choose the precedence of the subtraction operator relative to the existing operators. Justify your choice of precedence with two examples, one showing that your choice of precedence is "natural" to the programmer,

chosen precedence: _____ example: _____ reason: _____

and another example showing that a different choice of precedence could lead to counterintuitive semantics

bad precedence: _____ example: _____ reason: _____

Part 3: Associativity [6 points]

You must choose the associativity of the subtraction operator. Is it going to associate from the left or from the right? Justify your choice.

_____

_____

Part 4: Syntax-directed translation [7 points]

Extend the syntax-directed translation given below so that it supports the sub operator. The goal of the translation is to produce an NFA that accepts a string if and only if it matches the regular expression. You can use two functions in your actions:

- **intersection(N1,N2)**: returns a NFA that accepts a string iff it is accepted by both N1 and N2.
- **complement(N,S)**: returns a NFA N' that is a complement of N. That is, N' accepts a string s from S* only if s is not accepted by N. The set S is the alphabet, i.e., the set of characters we are interested in. The set S* is the set of all strings over that alphabet, including the empty string.

We will use the following functions in our actions, all of which do exactly what you would expect and as those we used in class:

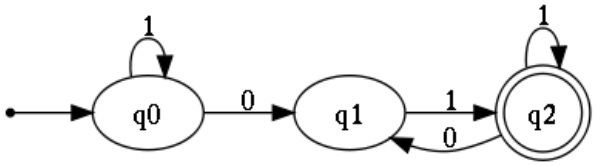
- **character(c)**: Returns a NFA that accepts the single character c.
- **star(R)**: Returns a NFA that accepts a string from L(R)*.
- **concatenation(R1,R2)**: Returns a NFA that accepts a string from L(R1) concatenated before any string from L(R2).
- **alternation(R1,R2)**: Returns a NFA that accepts a string iff it is accepted by either R1 or R2.

```
R ->  CHAR              %{ return character(n1.val) %}
 | R '*'    %dprec 1    %{ return star(n1.val) %}
 | R R      %dprec 2    %{ return concatenation(n1.val, n2.val) %}
 | R '|' R  %dprec 3    %{ return alternation(n1.val, n3.val) %}
 | '(' R ')'            %{ return n2.val %}

 ;

CHAR -> /[A-Za-z0-9]/ ;
```

Part 5: Implement *complement(N,S)* [7 points]

Show how a given an automaton N can be converted to its complement.

First, do this on an example. Rewrite the automaton below into its complement. The alphabet is S={0,1}.



Now show how the complement is computed in general given the nodes and transitions of N. Assume that N has nodes nodes(N), start node s, final nodes final(N) $\subseteq$ nodes(N) and transitions are given as the set T = { (n,m,c), … }, where is (n,m,c) denotes a transition from state n to state m on character c. You should use formal mathematical notation, although you may instead write a *short* description in English. As an example of the notation we expect,

transitions(concatenation(N1,N2)) = transitions(N1) $\cup$ transitions(N2) $\cup$ {(n1,start(N2),ϵ)|n1$\in$final(N1)}

nodes(N') = _____

start(N') = _____

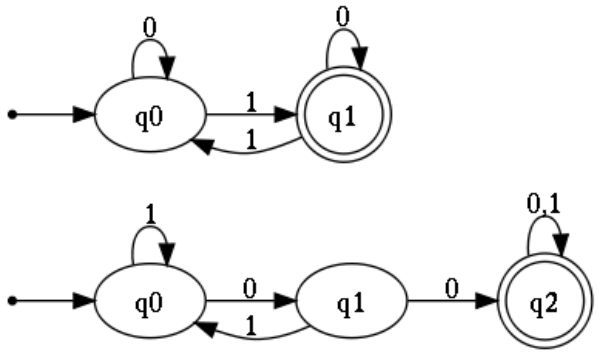
final(N') = _____

transitions(N') = _____

Part 6: Implement *intersection(N1,N2)* [7 points]

Recall that ***intersection(N1,N2)*** returns an NFA M that accepts a string iff it is accepted by both N1 and N2.

First, show the intersection of these two simple automata. Hint: if N1 has n states and N2 has m states, then the resulting automaton will need up to n*m states.



Now show how the automaton is computed in general:

nodes(M) = _____

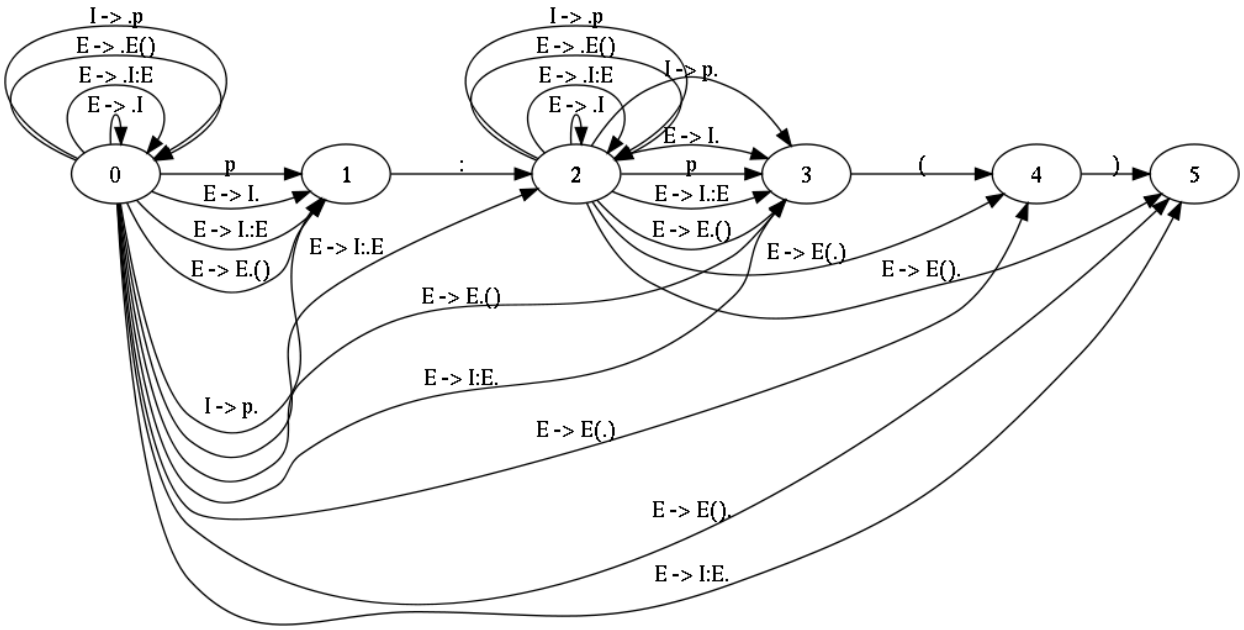
start(M) = _____

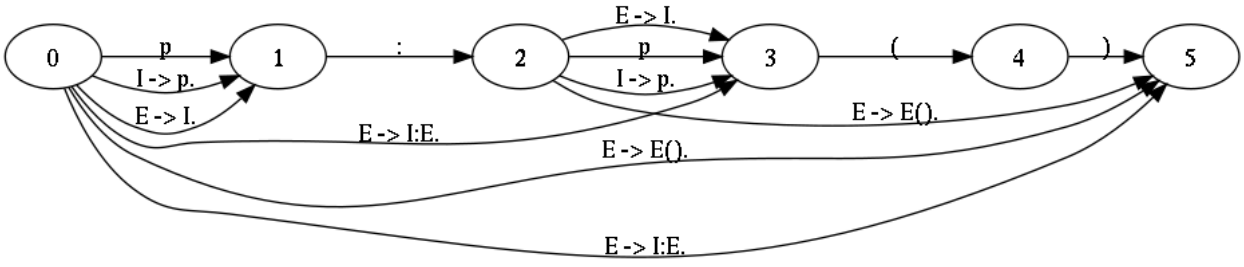
final(M) = _____

transitions(M) = _____

Problem 2: Early parser and parse trees [25 points]

The first figure shows the complete Early graph. The second figure shows the same graph but excludes all but completed edges.





Part 1 [8 points]

(Depending on how you think about parsing, you may find it beneficial to first answer Part 2).

In this subquestion, we are interested in whether, given an Earley parse graph such as the ones above, it is possible to determine what grammar was used by the Earley parser. Is it always possible or just sometimes possible (i.e., possible only for some inputs and for some grammars)? If it is not always possible, can you tell

from the graph whether a particular graph has enough information in it to reveal the *full* grammar? Answer these questions below.

Assume that the parse graph *g* is fully completed (i.e., no more Earley edges can be added), and that it is, of course, computed for a given input *s*.

Select from the choices in the sentence to make the sentence a true statement. Determining the exact grammar means that you give all the rules of the grammar and do not include rules that are not in the grammar.

1) Given an Earley parse graph *g*, it is <u>always</u> (i.e., for all parse graphs) possible to determine [the exact / a superset of the / a subset of the / nothing about the] grammar used by the parser.

Justify your answer:

_____

2) Given an Earley parse graph *g*, it is <u>sometimes</u> (i.e., for some parse graphs) possible to determine [the exact / a superset of the / a subset of the / nothing about the] grammar used by the parser. Give the most precise answer possible.

Justify your answer:

_____

Part 2 [8 points]

Write the grammar used in the parse graphs shown above.

Is your grammar complete (includes all the rules that were in the grammar given to the parser)? Circle one.

- yes
- no
- it is not possible to tell from the graph, but my grammar is a subset of the parser grammar
- it is not possible to tell from the graph, but my grammar is a superset of the parser grammar

Part 3 [9 points]

Does the parse graph shown above reveal ambiguity in the grammar?

yes no

Draw the parse tree(s) in the parse graph.

Draw a subtree that the Earley parser constructed but that did not end up in the final parse tree(s).

Problem 3: Adding language constructs with syntactic sugar [35 points]

In this problem, you will have to extend a simple language with a while statement.

Figure 1 shows complete Python code of an interpreter. The interpreter accepts an AST and evaluates it. Examine the code of the interpreter for a list of constructs that it supports. Figure 2 shows the complete grammar for a simple functional language. The parser translates a program in this functional language to an AST accepted by the interpreter in Figure 1.

Part 1: Scoping [7 points]

What kind of scoping does the interpreter implement? _____

Part 2: **Syntax-Directed Translation [7 points]**

Translate this program into an AST. Draw the AST the way Python interpreter would print it, as a **nested tuple-list structure**. Indent the AST nicely.

```
def foo(x,y) {
     x = x+y
     x
 }
 print foo(1,2)
```

AST:	***scratch space:***

Part 3: While implemented as a function [7 points]

Rewrite the following Python program in the simple functional language of Figures 1 and 2. You need to implement while as a function. You cannot simply rewrite `fact4` to be recursive. For this subquestion, you can modify neither Figure 1 nor Figure 2.

```
def fact4(n):
    fact = 1
    while (1<n):
        fact = fact * n
        n = n - 1
    return fact
```

Program:	***scratch space:***

Part 4 [6 points]

Which construct of the language is implemented with syntactic sugar?

_____

Where does the desugaring happen? Circle one. **Interpreter Parser**

Circle the lines of code in Figure 1 and/or 2 responsible for desugaring.

Part 5: While implemented with syntactic sugar [8 points]

Without modifying Figure 1 (the interpreter), add into the small language a while statement. After your extension, the following program must be correctly parsed and executed. You are allowed to change the parser; you are also allowed to create one or more library functions that will be distributed with the parser and the interpreter.

```
def fact4(n) {
    def fact = 1
    while (1<n) {
        fact = fact * n
        n = n - 1
    }
    fact
}
```

Change(s) to the parser: *(use back of pages for scratch space)*	***The library function(s).***

Figure 1. The interpreter.

```
def Exec(stmts):
    def eval(e,env):
        def lookup(name,env):
            if env == None: raise NameError("variable '%s' is
not defined" % name)
            try: return env[name]
            except: return lookup(name,env["__up__"])

        if (e[0] == 'int-lit'):   return e[1]
        if (e[0] == 'var'):       return lookup(e[1],env)
        if (e[0] == '*'):         return eval(e[1], env) *
eval(e[2], env)
        if (e[0] == '+'):         return eval(e[1], env) +
eval(e[2], env)
        if (e[0] == '-'):         return eval(e[1], env) -
eval(e[2], env)
        if (e[0] == '<'):         return eval(e[1], env) <
eval(e[2], env)
        if (e[0] == 'call'):      return apply(eval(e[1],env),
[eval(e,env) for e in e[2]], env)
        if (e[0] == 'lambda'):    return ((e[1],e[2]), env)  #
((arg list, body), env)
        if (e[0] == 'cond'):      return
[eval(e[2],env),eval(e[3],env)][not eval(e[1],env)] # This
evaluates both args and chooses one based on e[1]'s value.
        raise SyntaxError("Illegal AST node %s " % e)

    def execute(stmts,env):
        def update(name,env,val):
            if env == None: raise NameError("variable '%s' is
not defined" % name)
            if name in env: env[name] = val
            else: update(name,env["__up__"],val)

        val = None
        for s in stmts:
            if   s[0] == 'exp'  : val = eval(s[1],env)
            elif s[0] == 'def'  : val = eval(s[2],env);
env[s[1]] = val
            elif s[0] == 'asgn' : val = eval(s[2],env);
update(s[1],env,val)
            elif s[0] == 'print': print(eval(s[1],env))
```

```
        return val
    def apply(f, args, env):
        fargs = f[0][0]
        fbody = f[0][1]
        fenv  = f[1]
        scope = dict(zip(fargs, args))
        scope["__up__"] = fenv
        env = scope
        return execute(fbody, env)
    return execute(stmts, { "__up__": None})
```

Figure 2. The parser.

```
%ignore          /[ \t\v\f]+/
%optional NL /\n|;/ -- makes ';' and '\n' statement separators

%left  '<'
%left  '+'  '-'
%left  '*'

%%

P       -> S_list ;

S       -> E               %{ return ('exp', n1.val)%}
        |  Id '=' E        %{ return ('asgn', n1.val, n3.val) %}
        |  'def' Id '=' E  %{ return ('def', n2.val, n4.val) %}
        |  'def' Id '(' Id_list ')' '{' S_list '}'  %{ return
('def', n2.val, ('lambda', n4.val, n7.val)) %}
        |  'print' E       %{ return ('print', n2.val) %}
        ;
E       -> Prim
        |  BinE
        ;
BinE    -> E '+' E   %{ return ('+', n1.val, n3.val) %}
        |  E '-' E   %{ return ('-', n1.val, n3.val) %}
        |  E '<' E   %{ return ('<', n1.val, n3.val) %}
        |  E '*' E   %{ return ('*', n1.val, n3.val) %}
        ;
Prim    -> Number                  %{ return ('int-lit',
n1.val) %}
        |  Id                      %{ return ('var', n1.val) %}
```

```
        |  'lambda' '(' Id_list ')' '{' S_list '}' %dprec 1 %{
return ('lambda', n3.val, n6.val) %}
        |  'cond' '(' E ',' E ',' E ')'            %dprec 1 %{
return ('cond', n3.val, n5.val, n7.val) %}
        |  Prim '(' E_list ')'                     %dprec 2 %{
return ('call', n1.val, n3.val) %}
        |  '(' E ')'                 %{ return n2.val %}
        ;
Number  -> /[0-9]+/            %{ return int(n1.val) %} ;
Id      -> /[a-zA-Z_][a-zA-Z_0-9]*/ %{ return n1.val %} ;

S_list  -> _                    %{ return [] %}
        | S_ne_list
        ;
S_ne_list -> S                  %{ return [ n1.val ] %}
        |  S_list NL S          %{ return n1.val + [ n3.val ] %}
        ;
E_list  -> _                    %{ return [] %}
        | E_ne_list
        ;
E_ne_list -> E_ne_list ',' E  %{ return n1.val + [ n3.val ] %}
        |    E                %{ return [ n1.val ] %}
        ;
Id_list  ->   _                   %{ return [] %}
        |     Id_ne_list
        ;
Id_ne_list -> Id_ne_list ',' Id %{ return n1.val + [ n3.val ] %}
        |     Id                %{ return [ n1.val ] %}
        ;
```