

CS164 Second Midterm Exam

Fall 2014

November 24th, 2014

- Please read all instructions (including these) carefully.
- **This is a closed-book exam. You are allowed a one-page, one-sided handwritten cheat sheet.**
- Write your name and login on this first sheet, and your login at the top of each sheet.
- No electronic devices are allowed, including **cell phones** used merely as watches.
- Silence your cell phones and place them in your bag.
- 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 **todo** pages in this exam and **4** 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: _____

Problem	Max points	Points
1		
2		
3		
4		
Sub Total	100	

Question 1: Adding Constructs [15 points]

This question will test your understanding of how to add new language constructs.

Imagine you want to add a crazy new construct to the cs164 language. This construct will be called `untilNull`, and it will evaluate each expression in a sequence of 3 expressions until it reaches an expression that evaluates to null. It will not execute any expressions after that null-valued expression. If no expression in the list evaluates to null, this construct will evaluate all expressions. For the purposes of this question, we do not care what `untilNull` itself evaluates to.

Informal Syntax	Example 1	Output 1	Example 2	Output 2
<code>untilNull(E, E, E)</code>	<pre>def times(x, y){ def a = x*y print a if (a == 2){null} else{a} } untilNull(times(1,1), times(1,2), times(1,3))</pre>	<pre>1 2</pre>	<pre>def x = 0; def plus(num){ x = x+num null } untilNull(plus(1), plus(2), plus(3)) print x</pre>	<pre>1</pre>

Part A [5 points] Can the `untilNull` construct be implemented by extending the core cs164 language --- that is, by adding an `untilNull` AST node with `e1`, `e2`, and `e3` attributes and adding a new `untilNull` case to your interpreter's `evalExpression` function? If yes, write code for the new `untilNull` case. If no, write a one to two sentence description of why this design strategy is impossible. Write in only one box!

Yes	<pre>function evalExpression(node, env) { switch (node.type) { case 'untilNull': if (evalExpression(node.e1, env) !== null){ if (evalExpression(node.e2, env) !== null){ evalExpression(node.e3, env); } } return null; //we don't care about the return val } } }</pre>
No	

--	--

Part B [5 points] Can the `untilNull` construct be implemented by desugaring the `untilNull` construct into constructs already present in the cs164 language --- that is, by adding an `untilNull` AST node with `e1`, `e2`, and `e3` attributes and adding a new desugaring rule? If yes, write the desugaring rule. If no, write a one to two sentence description of why this design strategy is impossible. Write in only one box!

Yes	<pre> var DESUGAR_AST_RAW = { 'untilNull': 'lambda(){ \ if (%e1 != null) { \ if (%e2 != null) { \ %e3 \ }} \ }()' } </pre>
No	

Part C [5 points] Can the `untilNull` construct be implemented as a library function --- that is, by letting the parser produce a `call` node, and adding a function `untilNull` with arguments `e1`, `e2`, and `e3` defined in library.164? If yes, write code for the library function. If no, write a one to two sentence description of why this design strategy is impossible. Write in only one box!

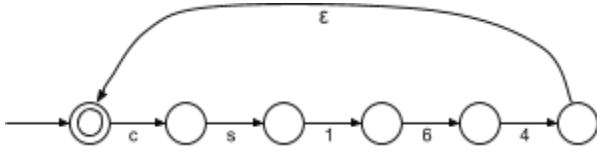
Yes	<pre> def untilNull(e1, e2, e3){ } </pre>
No	No. If we implement <code>untilNull</code> as a function, all three expressions will be evaluated every time the construct is used. Our spec requires that only the

	expressions up until and including the first null-valued expression be evaluated.
--	---

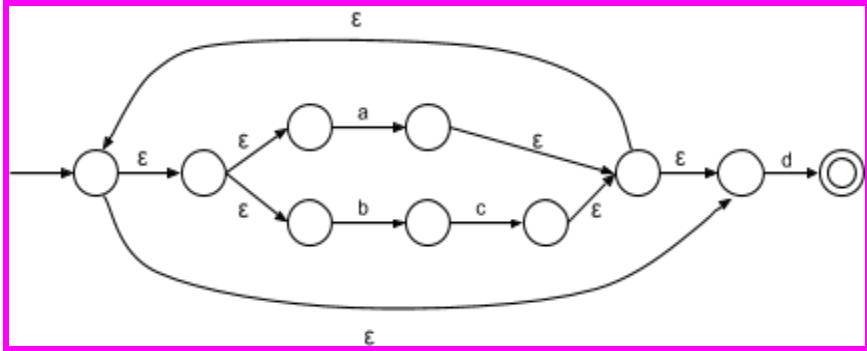
Question 2: Regular Expressions and Automata [30 points]

This question will test your understanding of regular expressions and automata.

Part A [2 points] Write a regular expression that accepts the same strings as the nondeterministic finite automaton (NFA) pictured below. Your regular expression may use concatenation, alternation ($|$), Kleene stars ($*$), and parentheses.

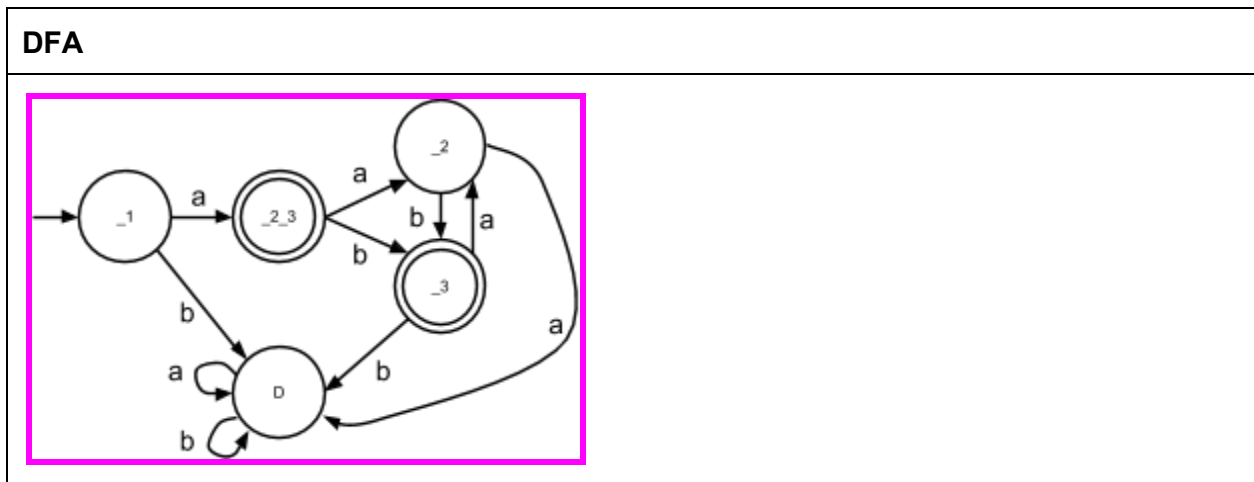
NFA	Regular Expression
	$(cs164)^*$

Part B [4 points] Draw an NFA that accepts the same strings as the following regular expression.

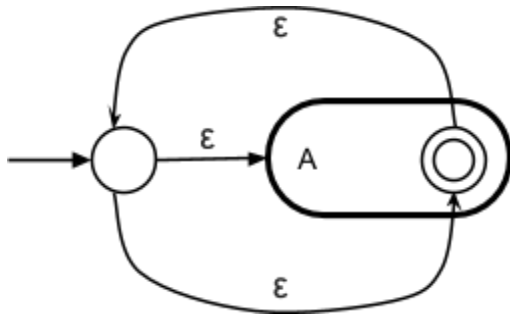
Regular Expression	NFA
$(a bc)^*d$	

Part C [6 points] Draw a state transition table (as seen in discussion section) for the following NFA, then a state transition table for a deterministic finite automaton (DFA) that accepts the same strings, then a DFA that accepts the same strings.

NFA	NFA state transition table	DFA state transition table																														
	<table border="1"> <thead> <tr> <th></th> <th>a</th> <th>b</th> </tr> </thead> <tbody> <tr> <th>1</th> <td>{2,3}</td> <td>∅</td> </tr> <tr> <th>2</th> <td>∅</td> <td>{3}</td> </tr> <tr> <th>3</th> <td>{2}</td> <td>∅</td> </tr> </tbody> </table>		a	b	1	{2,3}	∅	2	∅	{3}	3	{2}	∅	<table border="1"> <thead> <tr> <th></th> <th>a</th> <th>b</th> </tr> </thead> <tbody> <tr> <th>_1</th> <td>_2_3</td> <td>D</td> </tr> <tr> <th>_2_3</th> <td>_2</td> <td>_3</td> </tr> <tr> <th>_2</th> <td>D</td> <td>_3</td> </tr> <tr> <th>_3</th> <td>2</td> <td>D</td> </tr> <tr> <th>D</th> <td>D</td> <td>D</td> </tr> </tbody> </table>		a	b	_1	_2_3	D	_2_3	_2	_3	_2	D	_3	_3	2	D	D	D	D
	a	b																														
1	{2,3}	∅																														
2	∅	{3}																														
3	{2}	∅																														
	a	b																														
_1	_2_3	D																														
_2_3	_2	_3																														
_2	D	_3																														
_3	2	D																														
D	D	D																														



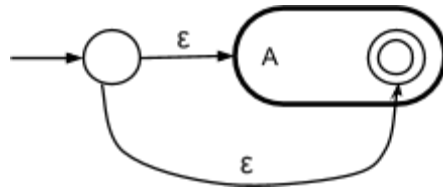
Part D [6 points] The following rule for converting the regular expression A^* into an automaton is not correct. The oval labeled “A” represents an NFA equivalent to the regular expression element A . Provide an example regular expression for which this rule will yield an NFA that is not equivalent to the regular expression; also provide a string that is accepted by either the regular expression or the NFA, but not the other.



Regular expression for which this rule fails:
 $(a(b)^*)^*$ _____

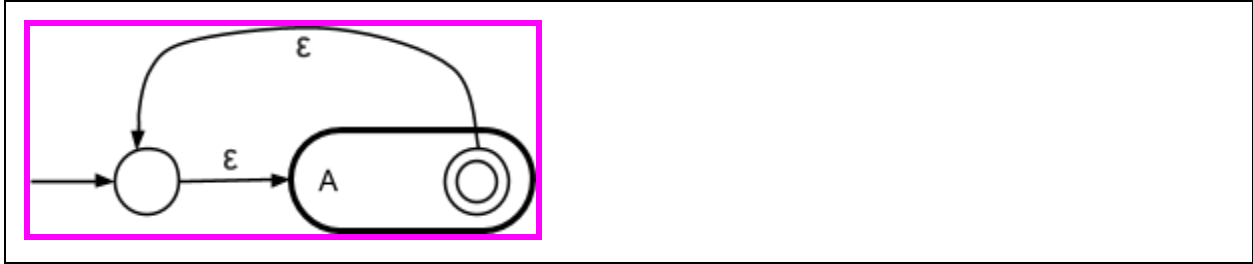
String accepted by either the above regular expression or its rule-generated automaton, but not both:
b _____

Part E [6 points] A question mark in a regular expression indicates that the regular expression accepts exactly 0 or 1 instances of the element that precedes the question mark. For instance, the regular expression **yay?** accepts the strings “ya” and “yay”. If our only other translation rules are the rules seen in class, is the following rule for converting **A?** to an automaton correct for all regular expressions? If yes, justify your answer. If no, provide an example regular expression on which it fails; also provide a string that is accepted by either the regular expression or the automaton, but not the other.



Yes, this rule works.	No, this rule doesn't work.
Justification: As long as we're using correct rules for all other constructs in regular expressions, this is ok, because there are never 'backward' edges leaving the final state. If we're using incorrect rules, like the Kleene star rule above, then this rule can fail, basically in the same way that the above Kleene star rule fails.	Regular expression for which this rule fails: _____ String accepted by either the above regular expression or its generated automaton, but not both: _____

Part F [6 points] Propose a rule for converting regular expressions with Kleene plus (+) into automata. Your rule should be presented as a drawing of the automaton to which **A+** should be converted, where **A** is any regular expression element. Label an oval with “A” to indicate that it is an NFA equivalent to the regular expression element **A**. A Kleene plus is like a Kleene star except that it requires at least one repetition of the preceding regular expression element.



Question 3: Grammars, SDT and Parsing [35 points]

This question will test your understanding of grammars, syntax directed translation and parsing. We provide a small grammar to express automata below. You will be fixing and adding onto this grammar.

```

%ignore      /[\t\v\f\r\n]+/

%%

S -> S Sep S
    | E
    ;

E -> Node Transition Node
    ;

Transition -> '->' Input
            ;

Node -> Id
      ;

Sep -> ';'
     ;

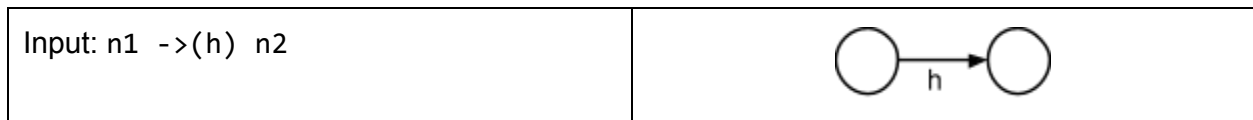
Input -> '(' Character ')'
        | _
        ;

Character -> /[a-zA-Z_0-9]/

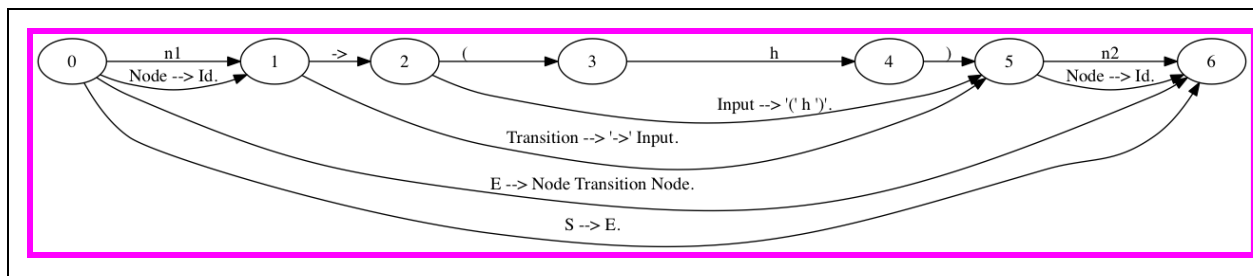
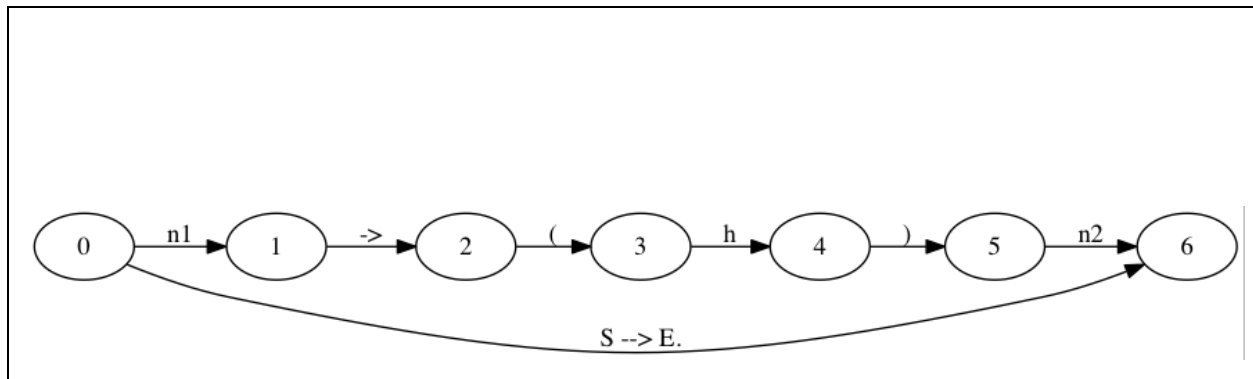
Id -> /[a-zA-Z_][a-zA-Z_0-9]+/ ;

```

Part A [10 points] The grammar allows simple automata to be easily specified. We provide an example of a string in the language, and the corresponding automata below.



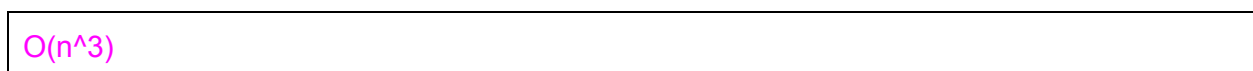
Draw the completed edges created by the Earley algorithm using the above grammar on the input “ $n1 \rightarrow (h) n2$ ”. We provide the final completed edge ($S \rightarrow E$) as an example of the labelling we expect.



Part B [4 points] The grammar above is ambiguous. Modify the grammar to remove ambiguity. You only need to write production(s) that are new or modified.



Part C [2 points] What is the worst-case running time of parsing an input string using the original (ambiguous) grammar?



Part D [3 points] Write an input which exhibits the worst case running time for the original grammar.

```
n1 ->(h) n2; n2 ->(i) n3; n3 ->(j) n4; ... ; n100 ->(a) n101
```

Part E [3 points] Now we want to modify the grammar to allow for paths. A path allows multiple adjacent transitions to be specified together. For example, the two following strings specify the same automaton.

```
n1 ->(h) n2; n2 ->(i) n3
```

```
n1 ->(h) n2 ->(i) n3
```

Again, you only need to write production(s) that are new or modified. Ensure that the new grammar is still unambiguous.

```
E -> E Transition Node
    | Node Transition Node
    ;
```

Part F [3 points] Now we want to modify the grammar to specify start states and accepting states. We give a few examples of the syntax below.

```
n1[start] ->(h) n2 ->(i) n3[accept]
n1[start, accept] ->(a) n1
```

Modify your grammar so that it accepts this new language. As before, make sure the grammar is unambiguous.

```
Node -> Id
      | Id '[' Modifier_list ']'
      ;

Modifier_list -> Modifier_list ',' Modifier
              | Modifier
              ;

Modifier -> 'start'
          | 'accept'
```

;

Part G [10 points] Complete the grammar below so that strings in the language are translated into the following function calls.

Function name	Description
<code>addEdge(nodeId, nodeId, input)</code>	Add a transition edge between two states. <code>input</code> should either be a string of length 1, specifying the input character or <code>null</code> , specifying an epsilon transition.
<code>addStart(nodeId)</code>	Specifies that <code>nodeId</code> is a start state.
<code>addAccept(nodeId)</code>	Specifies that <code>nodeId</code> is an accepting state.

We provide an example of translation below.

Input: `n1[start, accept] ->(c) n2 ->(s) n3 ->(1) n4 ->(6) n5 ->(4) n6 -> n1`

Function Call Trace	Automata
<pre>addStart("n1") addAccept("n1") addEdge("n1", "n2", "c") addEdge("n2", "n3", "s") addEdge("n3", "n4", "1") addEdge("n4", "n5", "6") addEdge("n5", "n6", "4") addEdge("n6", "n1", null)</pre>	

Note that the output represents a trace of functions called during the execution of the parser, not a textual program returned by parsing. Also, functions may be called in any order.

<code>%ignore</code>	<code>/[\t\v\f\r\n]+/</code>
<code>%%</code>	
<code>S -></code>	<code>S Sep E</code> %{ return %}
	<code> E</code> %{ return %}
	<code>;</code>

```

E -> E Transition Node
    %{ addEdge(n1.val, n3.val, n2.val); return n3.val %}
    | Node Transition Node
    %{ addEdge(n1.val, n3.val, n2.val); return n3.val %}
    ;

Transition -> '-' Input %{ return n2.val %}
    ;

Node -> Id
    %{ return n1.val %}
    | Id '[' Modifier_list ']'
    %{ if ('start' in n3.val)
        addStart(n1.val);
        if ('accept' in n3.val)
            addAccept(n1.val);
        return n1.val
    %}
    ;

Modifier_list -> Modifier_list ',' Modifier
    %{ var t = n1.val;
        t[n3.val] = true;
        return t
    %}
    | Modifier
    %{ var t = {};
        t[n1.val] = true;
        return t
    %}
    ;

Modifier -> 'start'
    | 'accept'
    ;

Input -> '(' Character ')' %{ return n2.val %}
    | _ %{ return null %}
    ;

Sep -> ';'
    ;

Character -> /[a-zA-Z_0-9]/

Id -> /[a-zA-Z_][a-zA-Z_0-9]+/ ;

```


Question 4: Bytecode translation [20 points]

In PA 4, you implemented compilation to bytecode of cs164 statements. In this question, we will consider an alternative protocol for bytecode generation, i.e. we are going to change how the recursive functions in the AST-to-bytecode compiler exchange information. We will adapt the compilation rules accordingly.

Recall the interface for the **btcnod**e function, that takes a node to translate, a target register, and the array of bytecode generated so far:

```
function btcnod(node, target, btc) { ... }
```

We would like to modify this interface such that **btcnod**e only takes one argument, the node to translate.

```
function btcnod(node) { ... }
```

We also require that **btcnod**e cannot access any global variables.

Part A [2 points] Give a data structure (a JavaScript object) that the new **btcnod**e function needs to return with this new protocol? Give a one-line description for each field you define.

- The new **btcnod**e will need to return a data structure with:
- **target**: the identifier of a fresh target node that it creates
 - **btc**: the bytecode array that the node translates to

Part B [8 points] Complete the bytecode translation code below for addition.

The input to **btcnode** for an addition is an AST whose nodes include the fields:

- **type**: the node type, i.e. **+**
- **operand1**: the AST for the first operand
- **operand2**: the AST for the second operand

The bytecode instruction for addition is an object with the following fields:

- **type**: the instruction type, **+**
- **operand1**: the register for the first operand
- **operand2**: the register for the second operand
- **target**: the target register that should receive the addition result

You may use the following library functions:

- **concat(xs1, xs2)** returns the concatenation of the two sequences **xs1** and **xs2**.
- **append(xs, x)** returns a new sequence that appends element **x** to the sequence **xs**.
- **uniquegen()** generates generate a unique name.

```
function btcnode(node) {
  switch (node.type) {
    case '+':
      var res1 = btcnode(node.operand1);
      var res2 = btcnode(node.operand2);
      var target = uniquegen();
      var addBtc = {
        'type': '+',
        'operand1': res1.target,
        'operand2': res2.target,
        'target': target
      };
      return {
        'btc': append(concat(res1.btc, res2.btc), addBtc),
        'target': target
      };
    break;
  }
}
```


Part C [10 points] Complete the bytecode translation below for function calls.

The input to **btcnnode** for a call is an AST which defines:

- **type**: the node type, i.e. **call**.
- **function**: the AST for the function
- **arguments**: the array of ASTs for the call arguments.

The bytecode instruction for the call is an object with the following fields:

- **type**: the instruction type, **call**
- **function**: the register that holds the function to call
- **arguments**: the array of registers that hold each argument value
- **target**: the target register that should receive the value returned by the call

You may use the library functions given for Part B.

Problem continued on the next page.

```
function btcnode(node) {
  switch (node.type) {
    case 'call':
      var btcs = [];
      var target = uniquegen();

      var fnRes = btcnode(node.function);

      var argTargets = [];
      var argBtcs = [];
      node.arguments.forEach(function(arg) {
        var argRes = btcnode(arg);
        argTargets = append(argTargets, argRes.target);
        btcs = concat(btcs, argRes.btc);
      });

      var callBtc = {
        'type': 'call',
        'function': fnRes.target,
        'arguments': argTargets,
        'target': target
      };
      return {
        'btc': append(concat(fnRes.btc, argBtcs), callBtc),
        'target': target
      };

      break;
    }
  }
}
```