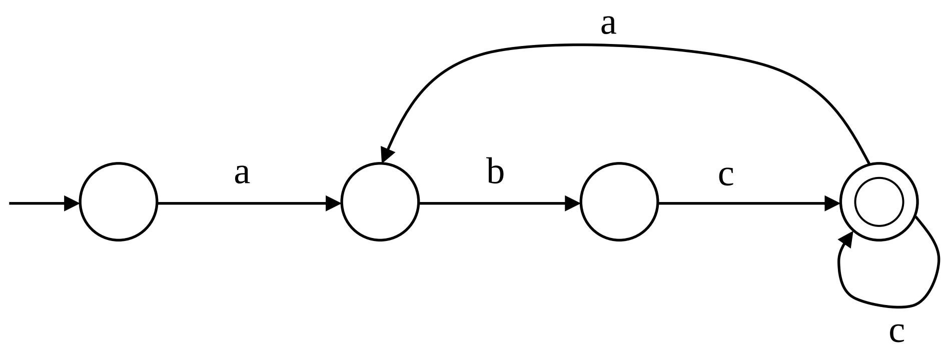
CSE 401 Midterm Exam Sample Solution 2/11/15

Question 1. (10 points) Regular expression warmup.

For regular expression questions, you must restrict yourself to the basic regular expression operations covered in class and on homework assignments: rs, r|s, r*, r+, r?, character classes like [a-cxy] and [^aeiou], abbreviations *name*=*regexp*, and parenthesized regular expressions. No additional operations that might be found in the "regexp" packages in various Unix programs, scanner generators like JFlex, or language libraries.

(a) (5 points) Write a regular expression that generates the set of strings recognized by this finite automaton:



(abc+)+

(b) (5 points) Write a regular expression that generates all non-empty strings of a's, b's, and c's where the first a precedes the first b if both a's and b's are present in a string.

One possible solution: c*a(a|b|c)* | (b|c)+

Question 2. (12 points) Regular expressions. Identifiers in the Ruby programming language are similar to those in many other languages, but a little different. An identifier:

- Contains any combination of letters, digits, and underscores (for this problem, assume that letters are only the ASCII characters `a-z` and `A-Z`, and digits are `0-9`)
- May not begin with a digit.
- May optionally have a single `$`, a single `@`, or two `@@` characters preceding the main part of the identifier.
- May optionally have a single `!`, a single `?`, or a single `=` character at the end.

Examples of valid identifiers: `abc123`, `@_Ab!`, `_` (a single underscore), `@@GLOB`, `x2=`.

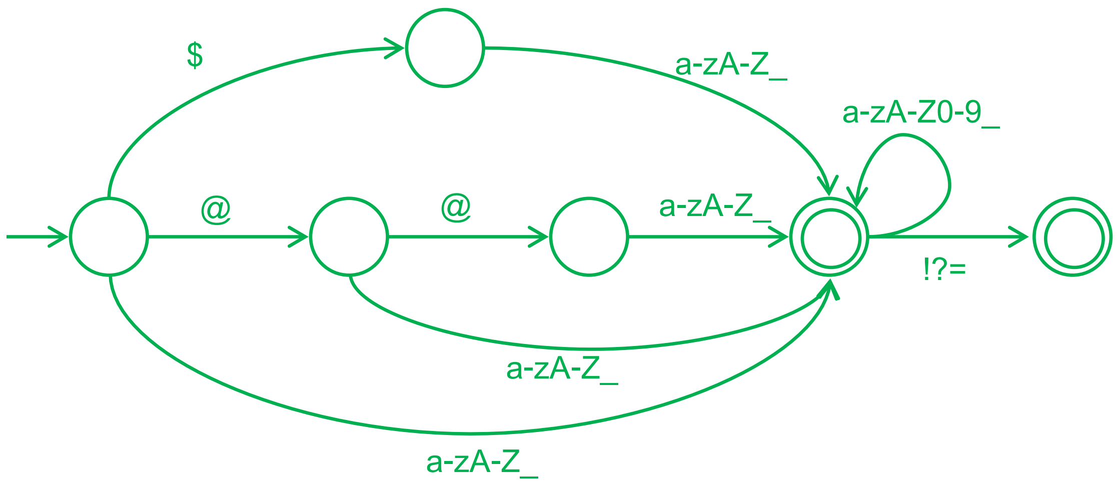
(a) (6 points) Give a regular expression that generates the set of valid Ruby identifiers.

letter = [a-zA-Z]

digit = [0-9]

($ | @@?)? (letter | _) (letter | digit | _)* (! | ? | =)?

(b) (6 points) Draw a DFA that accepts the set of valid Ruby identifiers. You only need to draw a DFA that is correct, you do not have to formally derive it from your answer to part (a) (although you're free to use a formal derivation if you'd like).



Question 3. (10 points) Scanners and tokens. Here is a small fragment found in a file:

```
a[k]++=1,234.5 #Hashtag //comment
ret/*we're done*/urn 17===42;
```

Below, list in order the tokens that would be returned by a scanner for the **full Java** programming language (not just the MiniJava subset) as it reads this input. If there is a *lexical* error in the input, indicate where the error is in the token stream and what is wrong, and continue to list the tokens corresponding to the remaining input, as would be done by a normal scanner. Remember that a scanner just detects lexical errors, and does not diagnose parsing or semantic errors detected by later parts of the compiler.

You can use any reasonable token names as long as your meaning is clear. The first three tokens are written for you:

ID(a) LBRACKET ID(k)

RBRACKET PLUSPLUS ASSIGN

NUMBER(1) COMMA NUMBER(234.5)

illegal character #

ID(Hashtag)

ID(ret) ID(urn) NUMBER(17)

EQUALEQUAL ASSIGN

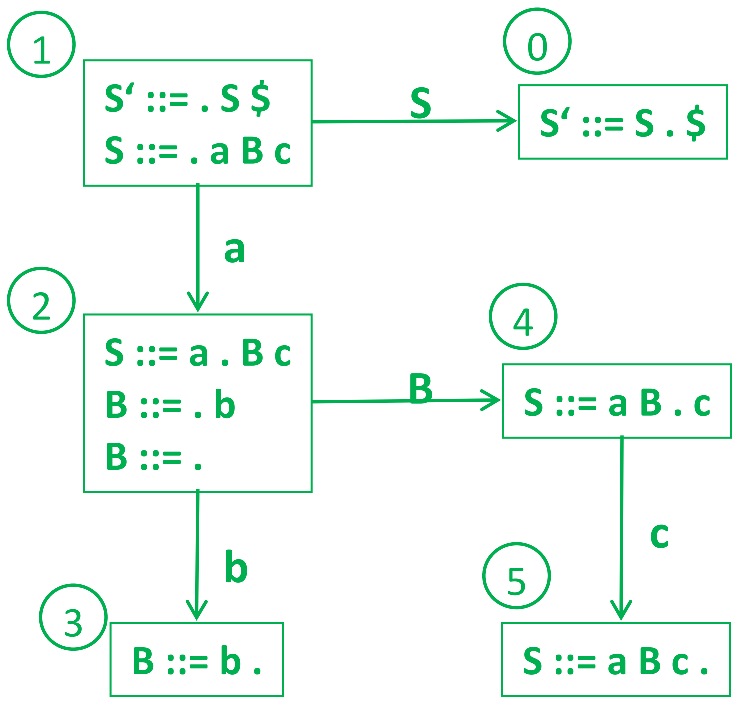
NUMBER(42) SEMICOLON

The most common errors on this question were because of trouble with the "principle of longest match" or else missing that the question was about tokens for full Java and not MiniJava. Many answers, for instance, listed two PLUS tokens for ++, but it is a single operator in Java.

Question 4. (34 point) The oh noz, not again, parsing question. Here is a tiny grammar.

0. *S′* ::= *S* $ ($ represents end-of-file)
1. *S* ::= a*B*c
2. *B* ::= b
3. *B* ::= ε

(a) (12 points) Draw the LR(0) state machine for this grammar.



The most common error here was treating ε as an input character and having a shift transition on ε from state 2 to a new state. But ε is not an actual input character to be matched – it indicates that *B* can derive the empty string.

(b) (6 points) Compute *nullable* and the FIRST and FOLLOW sets for the nonterminals *S* and *B* in the above grammar:

Symbol	nullable	FIRST	FOLLOW
S	**false**	**a**	**$**
B	**true**	**b**	**c**

(continued on next page)

Question 4. (cont.) Grammar repeated from previous page for reference.

0. $S' ::= S\ \$$
1. $S ::= \mathtt{a}B\mathtt{c}$
2. $B ::= \mathtt{b}$
3. $B ::= \varepsilon$

(c) (8 points) Write the LR(0) parse table for this grammar based on your LR(0) state machine in your answer to part (a).

	a	b	c	$	S	B
0				**acc**		
1	**s2**				**g0**	
2	**r3**	**r3,s3**	**r3**	**r3**		**g4**
3	**r2**	**r2**	**r2**	**r2**		
4			**s5**			
5	**r1**	**r1**	**r1**	**r1**		

(d) (2 points) Is this grammar LR(0)? Why or why not?

No. Shift-reduce conflict in state 2 on input b.

(e) (2 points) Is this grammar SLR? Why or why not?

Yes. Input b is not in FOLLOW(*B*) so state 2 should do s3 on input b, and that eliminates the shift-reduce conflict.

(f) (2 points) Is this grammar LL(1)? Why or why not?

Yes. The only issue is whether the two *B* productions have non-intersecting FIRST sets and they do not.

(g) (2 points) Everyone got 2 free points here. We realized as we were grading that parts (a)-(f) only added up to 32 and not the 34 points listed at the beginning of the question, so we added a 2-point part (g) and gave everyone full credit for it.

Question 5. (12 points) Concrete syntax. Full Java, as well as C and other languages, includes a three-operand conditional expression operator `?:`. An example of its use is in this assignment statement: `max = x>y ? x : y;` which stores the larger value of `x` or `y` in variable `max`. More generally, *e1*`?`*e2*`:`*e3* evaluates *e1*, then if *e1* is true, it evaluates *e2* and that is the value of the entire conditional expression. If *e1* is false, then *e3* is evaluated and it becomes the value of the conditional expression.

Suppose we have a language with the usual arithmetic expression grammar:

exp ::= *exp* + *term* | *exp* − *term* | *term*
term ::= *term* `*` *factor* | *term* `/` *factor* | *factor*
factor ::= *int* | *id* | `(` *exp* `)`

Now suppose we add the three-operand conditional expressions to this language by adding the following production to the ones above:

exp ::= *exp* `?` *exp* `:` *exp*

Show that the resulting grammar is ambiguous.

Here are two leftmost derivations of x?y:z+1, omitting most of the obvious steps.

exp => exp ? exp : exp =>* x ? y : exp => x ? y : exp + term =>* x ? y : z + 1.

exp => exp + term =>* x ? y : z + term =>* x ? y : z + 1

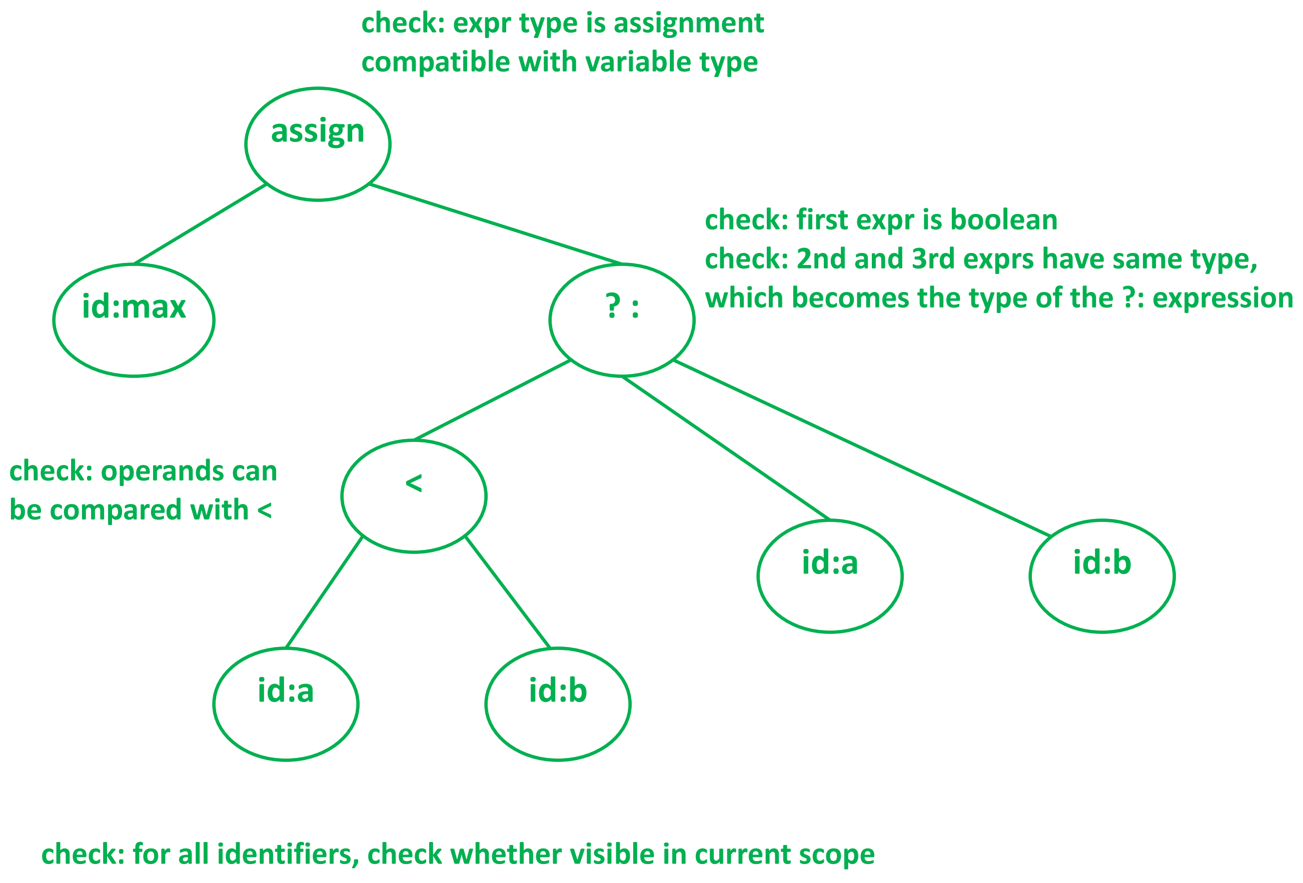
There are other solutions that show the grammar is ambiguous by generating other strings using two different leftmost or rightmost derivations. Another solution would be to show two different parse trees that generate the same string.

Question 6. (14 points) Abstract syntax and semantics. Let's assume that we've solved the ambiguity issues and have added conditional expressions from the previous problem to MiniJava. Suppose we encounter this statement in a program:

```
max = b < a ? a : b ;
```

(a) (6 points) Draw a tree below showing the abstract syntax for this statement. Don't worry about whether you match the AST classes in the MiniJava project code exactly (you're not expected to memorize that sort of thing). Just show nodes and edges of an AST that is appropriate for this expression.

(b) (8 points) After you've drawn the AST for this statement, annotate it by writing next to the appropriate nodes the checks or tests that need to be done in the static semantics / type-checking phase of the compiler to ensure that this statement does not contain any compile-time errors. You only need to indicate the necessary checks – you do not need to specify an attribute grammar, for example.


check: expr type is assignment compatible with variable type
assign
id:max
? :
check: first expr is boolean
check: 2nd and 3rd exprs have same type, which becomes the type of the ?: expression
check: operands can be compared with <
<
id:a
id:b
id:a
id:b
check: for all identifiers, check whether visible in current scope

Question 7. (8 points) The Ghost of LL Parsing Past. Consider the following grammar:

1. *S* ::= `a a`
2. *S* ::= `a b`

(a) (3 points) This grammar is not LL(1). Why not? (Give a concise technical reason)

FIRST(a a) and FIRST(a b) both contain a. So the intersection of the FIRST sets for the two right sides of the productions for *S* is not empty.

(b) (3 points) Write a grammar that generates the same language that is LL(1).

1. ***S* ::= a *S'***
2. ***S'* ::= a**
3. ***S'* ::= b**

(c) (2 points) Would it be possible to write a hand-coded recursive descent parser to recognize programs generated by the original grammar without rewriting it? Why or why not?

Yes. In a hand-written recursive-descent parser we can look ahead a few extra symbols when we need to, or we can sometimes defer deciding which production is being processed until later (the dangling else problem in an if-else statement is often handled this way).