CSE 401/M501 21au Midterm Exam 11/5/21 Sample Solution

Question 1. (18 points) We are designing a new programming language with the following conventions for method names. There are two types of method names: ordinary method names and private method names. Both sets of names can include combinations of lower-case letters and underscores only – no digits or upper-case letters. A name may not contain two or more consecutive underscore characters. An ordinary method name may not start with a leading underscore, but may contain a trailing underscore. A private method name *must* start with a single leading underscore and cannot end with an underscore. Examples of legal names:

Ordinary method names: `a_method` `another_method_` `simplename` `xy_`
Private method names: `_secret` `_private_method`

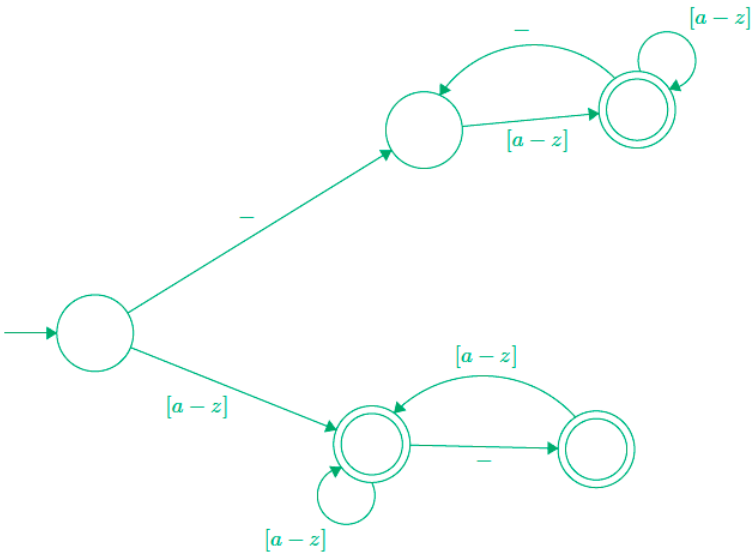
Fine print: You must restrict yourself to the basic regular expression operations covered in class and on homework assignments: *rs* , *r*|*s* , *r** , *r*+ , *r*?, character classes like `[a-cxy]` and `[^aeiou]`, abbreviations *name=regexp*, and parenthesized regular expressions. No additional operations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

(a) (8 points) Give a regular expression (possibly with subexpressions if it makes things easier) that generates all valid method names (ordinary and private) according to the above rules.

There are many possible solutions to both parts of this question. Here is one.

(_[a-z]+) + | ([a-z]+_?)+

(b) (10 points) Draw a DFA that accepts all valid method names (ordinary and private) according to the above rules.



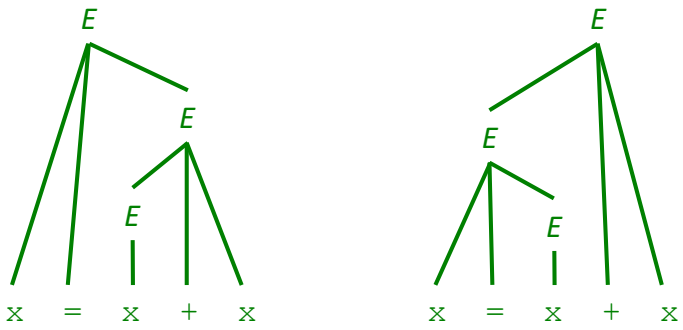
Question 2. (16 points) Ambiguity. Here is a small grammar for expressions involving assignment (=) and addition (+) and the terminal symbol x.

E ::= x = *E* | *E* + x | x

Is this grammar ambiguous? If so, give a proof that it is by showing two distinct parse trees or two different leftmost (or two different rightmost) derivations for some string generated by the grammar. If not, give an informal, but precise, argument why it is not ambiguous.

Note: whitespace in the grammar is only for readability and is not part of the grammar or the strings generated by it.

The grammar is ambiguous. Here are two different parse trees for x=x+x :



And here are two different leftmost derivations for x=x+x :

E => x = E => x = E + x => x = x + x
E => E + x => x = E + x => x = x + x

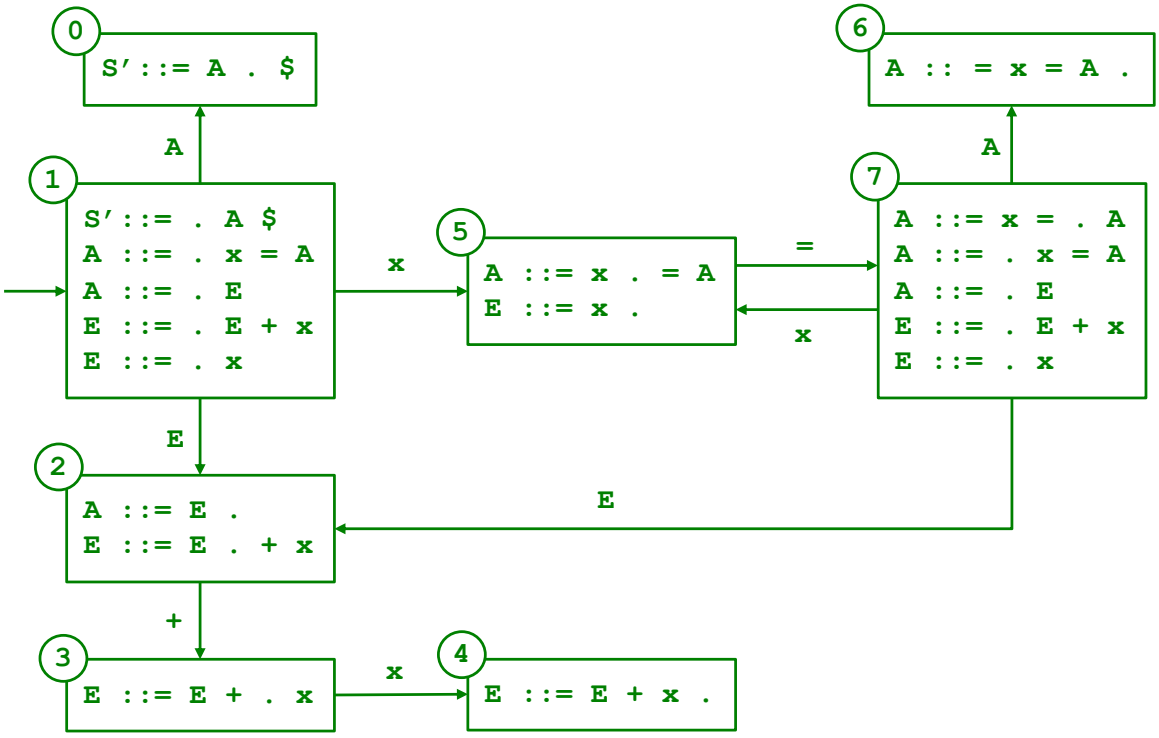
It turns out that in this case these are also rightmost derivations since there is only one non-terminal to expand at each step in the derivation. That would not be true in general.

Several answers confused the notions of derivations and parse trees. They are different, but related things. If a grammar is ambiguous, either can be used to show the ambiguity. Correct answers needed only to show one or the other. It was also fine to show distinct trees or derivations for a different string generated by the grammar.

Question 3. (32 points) The "OMG it's *back*!!" LR parsing question. Here is a different grammar for the language from the previous problem involving assignment (=) and addition (+) and the terminal symbol x. The extra *S′* ::= *A* $ rule needed to handle end-of-file in an LR parser has been added for you. As is usual, whitespace in the grammar is only for readability and is not part of the grammar or the strings generated by it.

0. *S′* ::= *A* $ ($ is end-of-file)
1. *A* ::= x = *A*
2. *A* ::= *E*
3. *E* ::= *E* + x
4. *E* ::= x

(a) (16 points) Draw the LR(0) state machine for this grammar. When you finish, you should number the states in the final diagram in whatever order you wish so you can use the state numbers to answer later parts of this question.



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

Symbol	nullable	FIRST	FOLLOW
A	no	x	$
E	no	x	+ $

Question 3. (cont.) Grammar repeated from previous page for reference:

0. S' ::= A $ ($ is end-of-file)
1. A ::= x = A
2. A ::= E
3. E ::= E + x
4. E ::= x

(c) (5 points) Is this grammar LR(0)? Explain why or why not. Your answer should describe **all** of the problems that exist if the grammar is not LR(0) by identifying the relevant state number(s) in your diagram in part (a) and the specific issue(s) in those state(s) (i.e., something like "state 47 has a shift-reduce conflict if the next input is x", but with, of course, state numbers and correct details from your diagram). If the grammar is LR(0) you should explain why. You do not need to write out the full LR(0) parse tables as part of your answer, just explain whether the state machine is LR(0) or not, and why.

It is not LR(0).

State 2 has a shift-reduce conflict if the next input symbol is + .

State 5 has a shift-reduce conflict if the next input symbol is = .

(d) (5 points) Is this grammar SLR? Explain why or why not. As with your answer to the previous part of the question, refer to states by number in the LR diagram in your answer to part (a) and give specific explanations of why the grammar is SLR or why not.

Yes it is.

FOLLOW(A) does not contain +. If we are in state 2 and the input is +, we should shift into state 3 and not reduce, which resolves the conflict.

For state 5, = is not in FOLLOW(E), so if the input is =, we should shift to state 7 and not reduce, which resolves that conflict.

Question 4. (14 points) LL parsing. Here is another look at the grammar from the previous question (the extra *S′* ::= *A* $ production needed to handle end-of-file in the LR parser has been omitted since it is not needed here).

1. *A* ::= x = *A*
2. *A* ::= *E*
3. *E* ::= *E* + x
4. *E* ::= x

Is this grammar, as written, suitable for constructing a top-down LL(1) predictive parser? If it is, your answer should give a technical explanation why it is. If not, your answer should give a technical explanation listing **all** of the problems with this particular grammar that prevent it from being suitable for a LL(1) predictive parser. You do not need to rewrite the grammar to fix the problems if there are any – just explain why it is or is not suitable for this use. (Hint: Explanations in terms of FIRST/FOLLOW and other properties needed by a LL(1) predictive parser may be helpful.)

No, it is not suitable for LL parsing.

For the pair of productions *A* ::= `x=`*A* and *A* ::= *E*, `x` is in FIRST(*E*) as well as in FIRST(`x=`*A*), so we cannot determine which production to use given one-symbol lookahead.

The same problem applies to the *E* ::= *E*`+x` and *E* ::= `x` productions. Terminal `x` is in FIRST(*E*) so this also violates the LL(1) requirement of disjoint first sets.

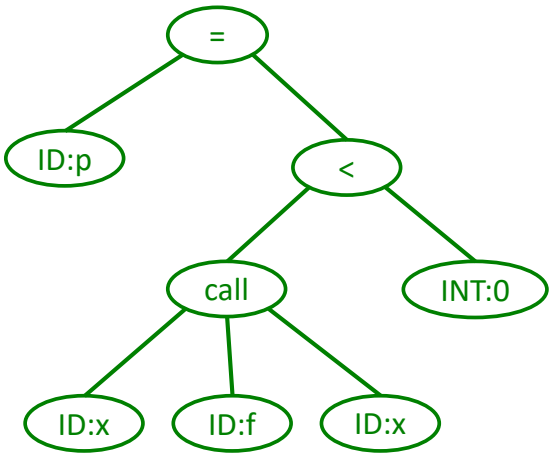
There are also potential problems with left recursion. We would need to factor the grammar to eliminate direct or potential indirect left recursions in the productions *A*::=*E* and *E*::=*E*`+x` to have a grammar that can be used directly for LL(1) parsing.

Question 5. (20 points) Semantics. Suppose we have the following assignment statement in a MiniJava program:

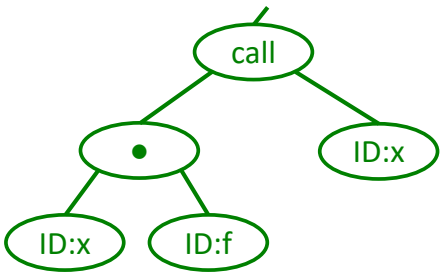
```
p = x.f(x)<0;
```

(a) (10 points) Draw an abstract syntax tree (AST) for this statement at the bottom of this page. You should use appropriate names for AST nodes and have an appropriate level of abstraction and structural detail similar to the AST nodes in the MiniJava project AST classes, but don't worry about matching the exact names of classes or nodes found in the MiniJava code.

(b) (10 points) Annotate your AST by writing next to the appropriate nodes the checks or tests that should be done in the static semantics/type-checking phase of the compiler to ensure that this assignment statement does not contain errors. You do not need to specify an attribute grammar – just indicate the necessary tests. If a particular test applies to multiple nodes, you can write it once and indicate which nodes it applies to, as long as your meaning is clear and readable. You may assume that `int` is the only numeric type in the language.



Here is another possible representation for the x.f(x) subtree (although different from the MiniJava AST node). The important thing is that it is a method call, not just selecting a field from an object.



Semantic checks needed, identified by node(s):

- **All identifier (ID:) nodes: verify that the identifier is declared and in scope.**
- **= (assignment): verify that the left-hand side operand (ID:p) designates an assignable location (lvalue); verify that the type of the expression (<) is assignment-compatible with the type of ID:p, which is Boolean here.**
- **<: verify that both operands have the same type or are comparable types (int here). Type of the < node is Boolean.**
- **Call: verify that the type of the expression `x` contains a field `f`, which is a method that has one parameter. The type of the method call parameter `x` must be the same as, or a subtype of, the declared parameter type of `f`. The result type of this node is the result type of `f` (int here)**

Note: The MiniJava AST method call node has three children: an expression, an identifier, which must be a method member of that expression's type, and the parameter expression(s) for the call. It's fine if your answer used something different as long as it is clear that it was a call of method x.f with parameter x.