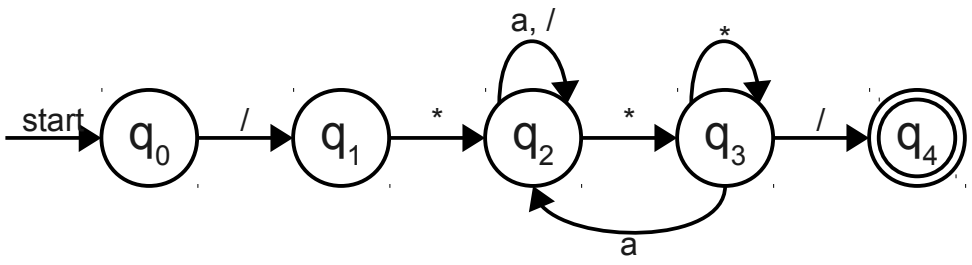
CS143 Practice Midterm Exam Solutions

Problem 1: C-Style Comments

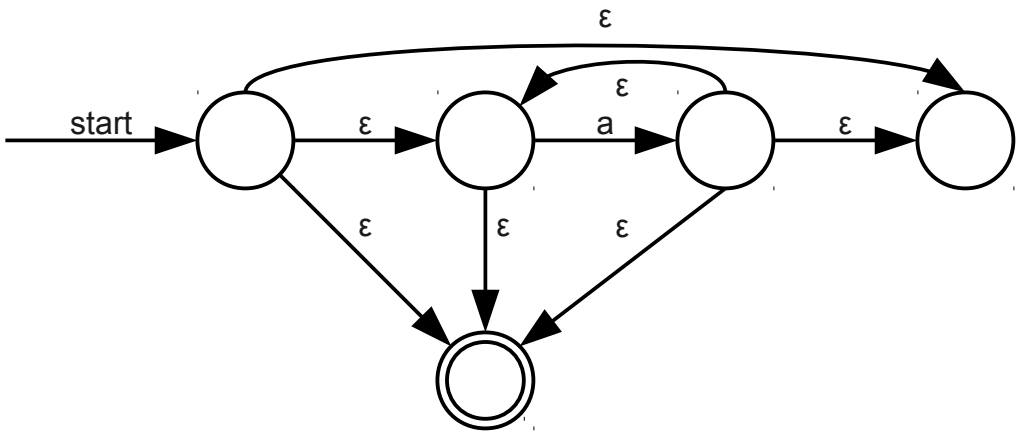
(i) One possible NFA is as follows:



The intuition behind this automaton is as follows. States q_0 and q_1 exist so that we match the start of the C-style comment. State q_2 represents being inside of the comment. State q_3 represents having matched the `*` at the beginning of the close-comment, and state q_4 represents having matched the end of the comment. Note that when we are inside a comment, if we read the sequence `*/`, we end up state q_4 and accept. We can't extend the comment any further, since if we tried we would try transitioning out of state q_4, which has no transitions defined.

The interesting detail is the transitions in state q_3. If we read a `*` in this state, then we stay put. This ensures that comments like `/******/` are handled correctly. If we read an non-star, non-slash character here, then we transition back to state q_2 to indicate that we didn't match the end-of-comment marker and need to try again.

(ii) This construction is very broken and there are many correct answers. For example, here is the automaton for **`(a*)!`**:



The problem with this automaton is that the regular expression **`(a*)!`** should not accept the empty string, since that's matched by **`a*`**. However, if we run this automaton on the empty string, we can begin at the start state and reach the accept state by an ε-transition. Consequently, this automaton would incorrectly say to accept the string when it ought to reject it.

More generally, if you have an automaton with states that have ε-transitions into the accept state, then this construction will not work because the states that have ε-transitions into the accept state will incorrectly transition into the new accept state, even though if they were ever reached in the original automaton they would cause the automaton to accept.

The automata suggested on the practice exam itself were incorrect. In fact, the automaton for **`a!`** would not match strings like **`b`** or **`ab`**.

Problem 2: LL(1) Parsing

(i) This grammar is not LL(1) for two reasons:

1. The production *Type* → *Type* **`*`** is left-recursive, and LL(1) cannot handle left-recursive grammars.
2. The productions *ArgList* → *Type* **`id`** **`,`***ArgList* and *ArgList* → *Type* **`id`** are left-factorable with a non-nullable prefix.

(ii) There are several solutions to this problem, but all of them involve eliminating the left recursion identified in (1) and left-factoring *ArgList*. Here is one possible way to do this:

(1) *Function* → *Type* **`id (`***Arguments***`)`**
(2) *Type* → **`id`** *Stars*
(3) *Stars* → **`*`***Stars*
(4) *Stars* → ε
(5) *Arguments* → *ArgList*
(6) *Arguments* → ε
(7) *ArgList* → *Type* **`id`** *MoreArgs*
(8) *MoreArgs* → **`,`** *ArgList*
(9) *MoreArgs* → ε

(iii) The answer to this question depends on what grammar you came up with in (ii). For our grammar, we have the following:

FIRST(*Function*) = {**`id`**}
FIRST(*Type*) = {**`id`**}
FIRST(*Stars*) = {**`ε`**, **`*`**}
FIRST(*Arguments*) = {**`ε`**, **`id`**}
FIRST(*ArgList*) = {**`id`**}
FIRST(*MoreArgs*) = {**`ε`**, **`","`**}

FOLLOW(*Function*) = {**`$`**}
FOLLOW(*Type*) = {**`id`**}
FOLLOW(*Stars*) = {**`id`**}
FOLLOW(*Arguments*) = {**`)`**}
FOLLOW(*ArgList*) = {**`)`**}
FOLLOW(*MoreArgs*) = {**`)`**}

(iv) The answer to this question depends on your answers to (ii) and (iii). For our grammar, the LL(1) parse table is as follows:

	id	*****	**(**	**)**	**,**	**$**
Function	1					
Type	2					
Stars	4	3				
Arguments	5			6		
ArgList	7					
MoreArgs				9	8	

Problem 3: LR Parsing

(i) For a grammar to be LR(0), every reduce item must be in its own state to avoid shift/reduce conflicts. This LR(0) automaton has several states that contain both shift and reduce items, so the grammar is not LR(0).

(ii) This grammar is not SLR(1). Note that FOLLOW(D) contains **id**, because D is followed by E, E can start with Q, and Q can start with **id**. This means that in state (1) there is a shift/reduce conflict, because we can't tell whether to shift **id** for the last item or reduce D to the empty string on **id**, which is contained in its FOLLOW set.

(iii) The LALR(1)-augmented grammar for the grammar is as follows:

$S_1 \rightarrow X_{1\text{-}2}$
$X_{1\text{-}2} \rightarrow D_{1\text{-}7}\ E_{7\text{-}11}$
$D_{1\text{-}7} \rightarrow P_{1\text{-}3}\ D_{3\text{-}4}$
$D_{1\text{-}7} \rightarrow \varepsilon$
$P_{1\text{-}3} \rightarrow$ **id id**
$D_{3\text{-}4} \rightarrow P_{3\text{-}3}\ D_{3\text{-}4}$
$D_{3\text{-}4} \rightarrow \varepsilon$
$P_{3\text{-}3} \rightarrow$ **id id**
$E_{7\text{-}11} \rightarrow Q_{7\text{-}12}\ E_{12\text{-}13}$
$E_{7\text{-}11} \rightarrow \varepsilon$
$Q_{7\text{-}12} \rightarrow$ **id = id**
$E_{12\text{-}13} \rightarrow Q_{12\text{-}12}\ E_{12\text{-}13}$
$E_{12\text{-}13} \rightarrow \varepsilon$
$Q_{12\text{-}12} \rightarrow$ **id = id**

(iv) The FOLLOW sets are as follows:

FOLLOW(S_1) = {**$**}
FOLLOW(X_{1-2}) = {**$**}
FOLLOW(D_{1-7}) = {**id**, **$**}
FOLLOW(P_{1-3}) = {**id**, **$**}
FOLLOW(D_{3-4}) = {**id**, **$**}
FOLLOW(P_{3-3}) = {**id**, **$**}
FOLLOW(E_{7-11}) = {**$**}
FOLLOW(Q_{7-12}) = {**id**, **$**}
FOLLOW(E_{12-13}) = {**$**}
FOLLOW(Q_{12-12}) = {**id**, **$**}

(v) The grammar is not LALR(1). Notice that the FOLLOW sets for the augmented nonterminals are all the same as the FOLLOW sets for the original nonterminals. Since FOLLOW sets are the union of all the LALR(1) lookahead sets for a given nonterminal, this means that the LALR(1) lookaheads for all the items are the same as the SLR(1) lookaheads. Since the grammar is not SLR(1), it therefore isn't LALR(1) either.

Alternatively, you could show that the LALR(1) grammar has a shift/reduce conflict in it. For example, in the first state, there is a shift item from P → ·**id id** that says to shift on **id**. However, we also should do a reduction D → ε on FOLLOW(D_{1-7}), which contains **id**. Since we have a shift/reduce conflict, the grammar is not LALR(1).

(vi) This grammar contains a shift/reduce conflict in its LALR(1) table. Since merging LR(1) states with identical cores can only introduce reduce/reduce conflicts, and never shift/reduce conflicts, this shift/reduce conflict must also be present in the LR(1) parse table. Consequently, this grammar cannot be LR(1).

You also could have constructed a piece of the LR(1) automaton for the grammar and discovered the shift/reduce conflict explicitly, though I think the above line of reasoning is much simpler.

Interesting note: This question was based off of a tricky detail in the second programming assignment that often caused inexplicable shift/reduce conflicts that wouldn't go away easily with precedence declarations. Part of the Decaf grammar looks like this:

StmtBlock → **{** *VarDecl* Stmt** **}**

Since the * symbol is an extension to context-free grammars, part of the assignment was to figure out how to rewrite these symbols as nonterminals. A common way of doing this was as

*VarDecl** → *VarDecl VarDecl** | ε
*Stmt** → *Stmt Stmt** | ε

Moreover, one way of expanding out *VarDecl* is as

VarDecl → **`id id;`**

and one way of expanding statement looks like

Stmt → **`id = id;`**

If you look at the grammar for this question, you'll notice that it's pretty much identically the above grammar, except with different symbols. The answers to this question pin down precisely why this conflict exists and why it's so hard to eliminate – the grammar isn't ambiguous; it's just not LR(1).

To fix this, one trick is to rewrite the above productions as

*VarDecl** → *VarDecl* VarDecl* | ε
*Stmt** → *Stmt Stmt** | ε

This produces exactly the same strings as the original grammar, but ends up being both LR(1). **`bison`** can indeed handle this grammar.

Hope this clarified things!

Problem 4: Right-to-Left Parsing

(i) Yes, there are many grammars that are RR(1) but not LL(1). Here's a simple example:

S → A
A → ab | a

This grammar is not LL(1) because there is a FIRST/FIRST conflict between the two productions of A. However, this grammar is indeed RR(1) because it can be parsed with this RR(1) parsing table:

	a	b	$
S	S → A		
A	A → a	A → ab	

(ii) Yes, there are many grammars that are RL(0) but not LR(0). In fact the previous grammar meets this criterion as well:

$S \rightarrow A$
$A \rightarrow ab \mid a$

Here are the RL(0) configurating sets:

(1) $S \rightarrow A\cdot$
$A \rightarrow ab\cdot$
$A \rightarrow a\cdot$

(2) $A \rightarrow a\cdot b$

(3) $A \rightarrow \cdot a$

(4) $A \rightarrow \cdot ab$

(5) $S \rightarrow \cdot A$

Since each configurating set contains at most one reduce item, the grammar is RL(0). However, it is not LR(0), since the configurating sets are

(1) $S \rightarrow \cdot A$
$A \rightarrow \cdot ab$
$A \rightarrow \cdot a$

(2) $A \rightarrow a\cdot$
$A \rightarrow a\cdot b$

(3) $A \rightarrow ab\cdot$

(4) $S \rightarrow A\cdot$

Notice that there is a shift/reduce conflict in state (2).

(iii) No grammars can be parsed with a reverse Earley parser but not an Earley parser, since an Earley parser works on every CFG.

(iv) No grammar meets these three properties. If the grammar were to produce two different parse trees for some string, it would have to be ambiguous. However, no ambiguous grammar is LR(0).