

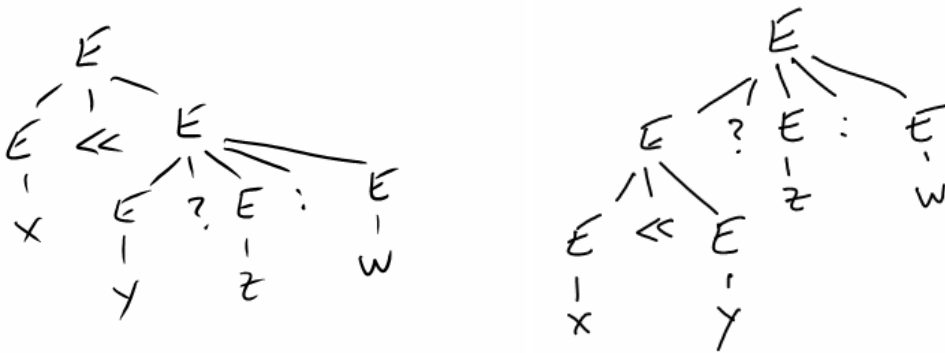
1. Here is a grammar for part of C expressions:

$$E \rightarrow id \mid E ? E : E \mid E \ll E$$

This grammar is ambiguous.

(a) Give a sentence that has two distinct parse trees, and show those parse trees.

x << y ? z : w



(b) Explain how you might use ocamlc associativity declarations to resolve the conflict caused by this ambiguity. Be specific about what declarations you would use and what their effect would be. (The actual rules for resolving this ambiguity in C or Java do not matter; you just need to use the rules to resolve it in some way.)

When we have << on the stack and ? in the input, if we reduce instead of shift, we'll get the tree on the right. We can do this by giving precedence of << over ?, which we do by putting an associativity declaration for ? before one for <<:

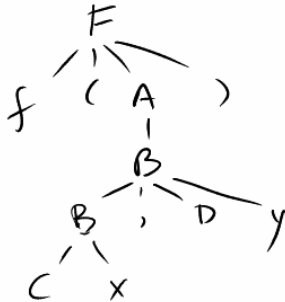
```
%nonassoc ?  
%left <<
```

(Alternatively, we could reverse the order of these declarations and get the parse on the left.)

2. Consider this grammar:

$F \rightarrow id (A)$
 $A \rightarrow \varepsilon \mid B$
 $B \rightarrow id id \mid B, id id$

(a) Show a parse tree for the following sentence: $f(C x, D y)$



(b) Write down all the sentential forms in that parse tree.

F f(A) f(B) f(B, D y) f(C x, D y)

(c) Calculate FIRST(F), FIRST(A), and FIRST(B).

FIRST(F) = {id} FIRST(A) = { id, • } FIRST(B) = {id}

(d) Calculate FOLLOW(F), FOLLOW(A), and FOLLOW(B) (assuming eof is in FOLLOW(F)).

FOLLOW(F) = {eof} FOLLOW(A) = {) } FOLLOW(B) = { }, ', ' }

(e) Why is this not an LL(1) grammar?

It is left-recursive. (In addition, the two productions from B do not satisfy the LL(1) condition, in that FIRST(id id) and FIRST(B id id) are not disjoint.)

(f) Transform it to an equivalent LL(1) grammar, and write a recursive descent parser for it. Assume the token type is

token = Id | LParen | RParen | Comma

and the parseF function has type

parseF: token list -> (token list) option

where option is the type defined as: `type 'a option = Some 'a | None.`

```
F -> id ( A )
A -> ε | id id B
B -> ε | , id id B
```

Note that FOLLOW(A) = FOLLOW(B) = {) }, FIRST(A) = { id, • }, and FIRST(B) = { •, ','}, so this satisfies the LL(1) condition.

```
let rec parseF lis = match lis with
  Id::LParen::lis' -> (match parseA lis' with
    Some (RParen :: lis'') -> Some lis''
  | _ -> None)
| _ -> None
```

```
let rec parseA lis = match lis with
  Id::Id::lis' -> parseB lis'
| _ -> Some lis (* could check for RParen here *)
```

```
let rec parseB lis = match lis with
  Comma::Id::Id::lis' -> parseB lis'
| _ -> Some lis (* could check for RParen here *)
```

3. Consider this grammar:

```
E -> E + T | T
T -> T * id | id
```

Show a complete shift-reduce parse for the sentence $x+y*z$. Include columns Action, Stack, and Input, as we did in class. (Warning: this may take more than one piece of paper, depending upon how big you write. There are 11 steps, counting the final “Accept” step.)

Action	Stack	Input
Sh		X+y*z
RT → id	X	+y*z
RE → T	T X	+y*z
Sh	E T X	+y*z
Sh	E + T X	y*z
RT → id	E + y T X	*z

Action	Stack	Input
Sh	E + T T X	*z
Sh	E + T * T X	z
RT → T → id	E + T * z T X	
RE → E + T	E + T * z T X	
Acc	E + T * z T X	

4. This grammar is not LR(1). It has a shift-reduce conflict:

A → B, \$
B → id | id, B

Show parse trees for two sentences with the following property: they lead to the same stack configuration (meaning, the roots of the trees on the stack are the same), with the same lookahead symbol, but in one case the correct action is the shift and in the other it is to reduce. Show that stack configuration.

