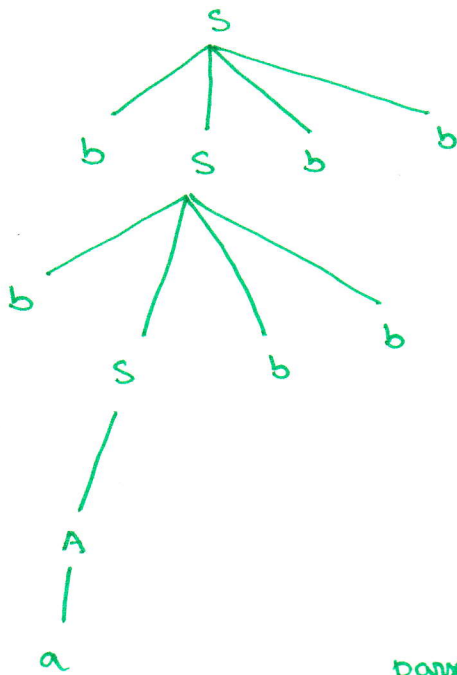


1. $S ::= bSbb \mid A$

$A ::= aA \mid \epsilon$

Example: $bbabbbb$

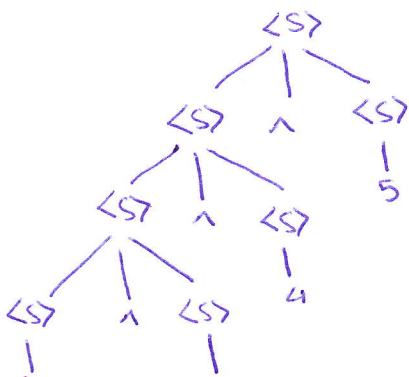
$S \Rightarrow bSbb \Rightarrow bbSbbbb \Rightarrow bbAbbbb \Rightarrow bba bbbb$



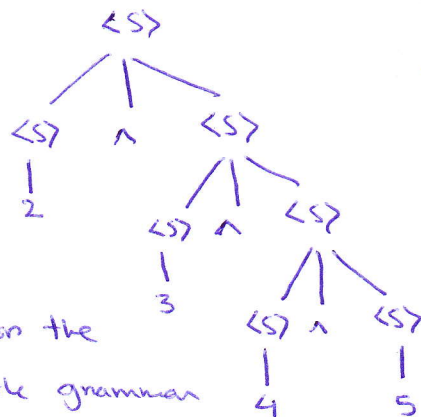
parse tree

2. (a) $2^3 3^4 5$

$\langle S \rangle \Rightarrow \langle S \rangle^{\wedge} \langle S \rangle$
 $\Rightarrow \langle S \rangle^{\wedge} \langle S \rangle^{\wedge} \langle S \rangle$
 $\Rightarrow \langle S \rangle^{\wedge} \langle S \rangle^{\wedge} \langle S \rangle^{\wedge} \langle S \rangle$
 $\Rightarrow 2^{\wedge} \langle S \rangle^{\wedge} \langle S \rangle^{\wedge} \langle S \rangle$
 $\Rightarrow 2^{\wedge} 3^{\wedge} 4^{\wedge} 5$



$\langle S \rangle \Rightarrow \langle S \rangle^{\wedge} \langle S \rangle$
 $\Rightarrow \langle S \rangle^{\wedge} \langle S \rangle^{\wedge} \langle S \rangle$
 $\Rightarrow \langle S \rangle^{\wedge} \langle S \rangle^{\wedge} \langle S \rangle^{\wedge} \langle S \rangle$
 $\Rightarrow 2^{\wedge} 3^{\wedge} 4^{\wedge} 5$



Different parse tree for the same sentence. So the grammar is ambiguous.

(b) As it is ambiguous it is not LL(1)

(c) $\langle S \rangle ::= \langle \text{Num} \rangle \langle A \rangle$

$\langle A \rangle ::= \wedge \langle S \rangle \mid \epsilon$

$\langle \text{Num} \rangle ::= 1|2|3|4|5$

(d,e)

$\text{First}(\langle S \rangle) = \{1, 2, 3, 4, 5\}$

$\text{First}(\langle A \rangle) = \{\wedge, \epsilon\}$

$\text{First}(\langle \text{Num} \rangle) = \{1, 2, 3, 4, 5\}$

$\text{Follow}(\langle S \rangle) = \{\$\}$

$\text{Follow}(\langle A \rangle) = \{\$\}$

$\text{Follow}(\langle \text{Num} \rangle) = \{\wedge, \$\}$

Parse Table

	1	2	3	4	5	\wedge	$\$$
$\langle S \rangle$	$S \rightarrow \text{Num } A$	$S \rightarrow \text{Num } A$	$S \rightarrow \text{Num } A$	$S \rightarrow \text{Num } A$	$S \rightarrow \text{Num } A$		
$\langle A \rangle$						$A \rightarrow \wedge S$	$A \rightarrow \epsilon$
$\langle \text{Num} \rangle$	$\text{Num} \rightarrow 1$	$\text{Num} \rightarrow 2$	$\text{Num} \rightarrow 3$	$\text{Num} \rightarrow 4$	$\text{Num} \rightarrow 5$		

Then LL(1) and unambiguous

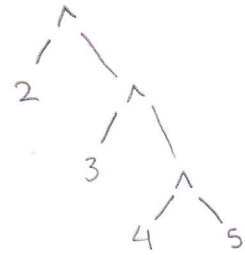
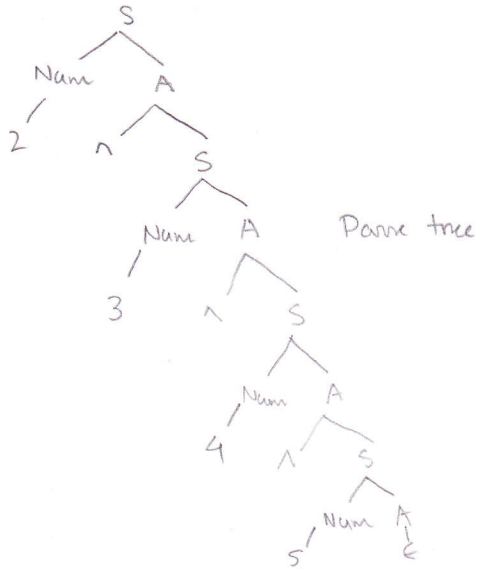
2. (f) $2^3^4^5$

$\langle S \rangle \Rightarrow \langle \text{Num} \rangle \langle A \rangle \Rightarrow \langle \text{Num} \rangle \wedge \langle S \rangle \Rightarrow \langle \text{Num} \rangle \wedge \langle \text{Num} \rangle \langle A \rangle \Rightarrow \langle \text{Num} \rangle \wedge \langle \text{Num} \rangle \wedge \langle S \rangle$

$\Rightarrow \langle \text{Num} \rangle \wedge \langle \text{Num} \rangle \wedge \langle \text{Num} \rangle \langle A \rangle \Rightarrow \langle \text{Num} \rangle \wedge \langle \text{Num} \rangle \wedge \langle \text{Num} \rangle \wedge \langle S \rangle$

$\Rightarrow \langle \text{Num} \rangle \wedge \langle \text{Num} \rangle \wedge \langle \text{Num} \rangle \wedge \langle \text{Num} \rangle \langle A \rangle \Rightarrow \langle \text{Num} \rangle \wedge \langle \text{Num} \rangle \wedge \langle \text{Num} \rangle \wedge \langle \text{Num} \rangle$

*
 $\Rightarrow 2^3^4^5$



Syntax tree

3(b). Write a grammar that can handle normal expression like "3+6*2"

Operator: + - * /

digit: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Easy solution:

$$\begin{aligned} \langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \\ &| \langle \text{expr} \rangle - \langle \text{expr} \rangle \\ &| \langle \text{expr} \rangle * \langle \text{expr} \rangle \\ &| \langle \text{expr} \rangle / \langle \text{expr} \rangle \\ &| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 \end{aligned}$$

Problem: This grammar is ambiguous, cannot handle the precedence of operators like

$$3+6*2 \quad \left[\begin{array}{l} (3+6)*2 \\ 3+(6*2) \end{array} \right.$$

'*' '/' have higher precedence than '+' '-'.

We have to rewrite the grammar that can enforce that

$$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{expr} \rangle - \langle \text{term} \rangle \mid \langle \text{expr} \rangle \langle \text{term} \rangle$$

$$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{term} \rangle / \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$$

$$\langle \text{factor} \rangle \rightarrow \text{digit} \mid (\langle \text{expr} \rangle)$$

Operator: '\$\$' '||' '!' (boolean op)

< <= > >= == (comp op)

= (assignment op)

+ - * / (arithmetic op)

Easy Grammar:

<whilestmt> → WHILE (<exp>) <stmt>

<expn> → <exp> \$\$ <exp>

| <exp> || <exp>

| ! <exp>

| <exp> < <exp>

| <exp> <= <exp>

| <exp> > <exp>

| <exp> >= <exp>

| <exp> == <exp>

| <exp> = <exp>

| <exp> + <exp>

| <exp> - <exp>

| <exp> * <exp>

| <exp> ./ <exp>

| (<exp>)

| ID

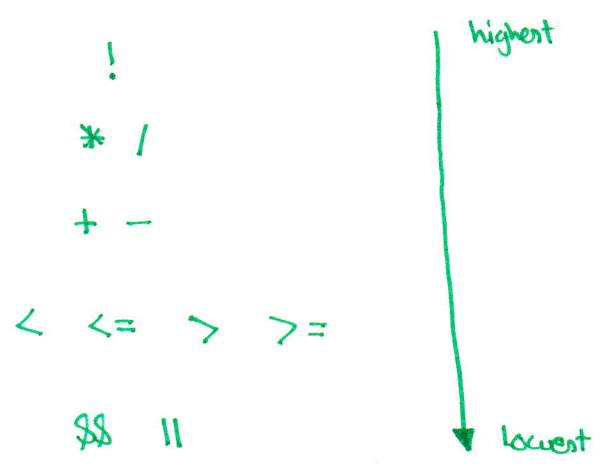
| VALUE

<stmt> → 'j' | <exp>; | { <stmt> } | <whilestmt>

<stmt> → ε | <stmt> <stmt>

Is it ambiguous? Does the precedence and associativity handled properly?

The precedence of operator in C



<exp> → ID = ⁼ <exp0> | <exp0>

<exp0> → <exp0> \$\$ <exp1> | <exp0> || <exp1> | <exp1>

<exp1> → <exp1> < <exp2> | <exp1> <= <exp2> | <exp1> > <exp2>

| <exp1> >= <exp2> | <exp2>

<exp2> → <exp2> + <exp3> | <exp2> - <exp3> | <exp3>

<exp3> → <exp3> * <exp4> | <exp3> / <exp4> | <exp4>

<exp4> → ! <exp5> | <exp5>

$\langle \text{exprs} \rangle \rightarrow (\langle \text{exp} \rangle)$

| ID

| VALUE

The rest are similar to the easy CFG.

3(a)

$\langle \text{var-decl} \rangle \rightarrow \langle \text{type} \rangle \langle \text{var-decl}' \rangle ;$

$\langle \text{var-decl}' \rangle \rightarrow \langle \text{var-decl}'' \rangle | \langle \text{var-decl}' \rangle, \langle \text{var-decl}'' \rangle$

$\langle \text{var-decl}'' \rangle \rightarrow \langle \text{var-decl-ID} \rangle | \langle \text{var-decl-id} \rangle = \langle \text{var-initializer} \rangle$

$\langle \text{var-decl-id} \rangle \rightarrow \text{ID} | * \langle \text{var-decl-id} \rangle | \langle \text{var-decl-id} \rangle [\text{IVAL}]$

$\langle \text{var-initializer} \rangle \rightarrow \text{IVAL} | \text{FVAL}$

$\langle \text{type} \rangle \rightarrow \text{INT} | \text{FLOAT}$

`int a=5;`

$\langle \text{var-decl} \rangle \Rightarrow \langle \text{type} \rangle \langle \text{var-decl}' \rangle ;$

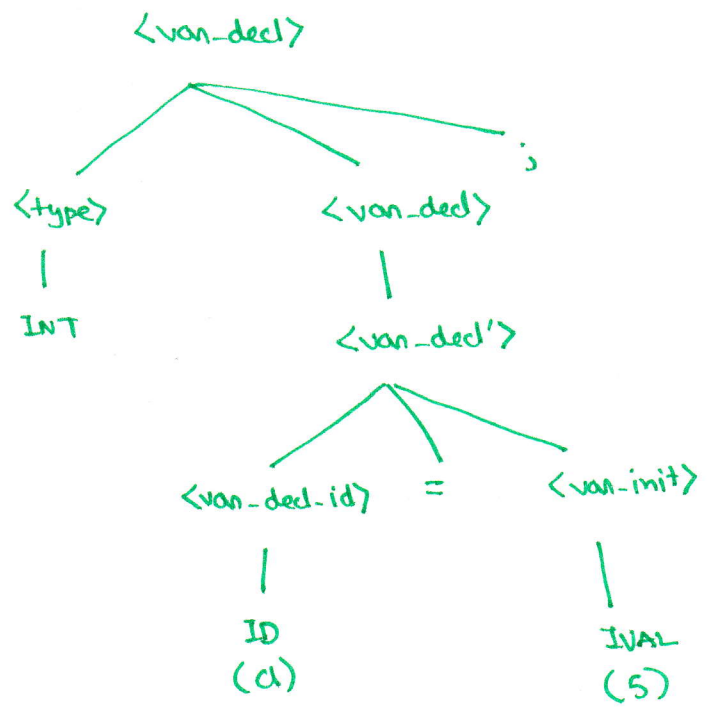
$\Rightarrow \langle \text{type} \rangle \langle \text{var-decl}' \rangle ;$

$\Rightarrow \langle \text{type} \rangle \langle \text{var-decl-id} \rangle = \langle \text{var-init} \rangle ;$

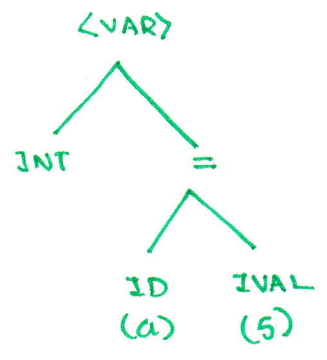
$\Rightarrow \langle \text{type} \rangle \text{ID} = \langle \text{var-init} \rangle ;$

$\Rightarrow \text{INT ID} = \langle \text{var-init} \rangle ;$

$\Rightarrow \text{INT ID} = \text{IVAL} ;$



parse tree



Syntax tree

4. (a) $L ::= Aa | Bb$
 $A ::= Lb | aa$
 $B ::= bBb | ba$

Simplifying \Rightarrow

$L ::= Lba | aaa | Bb$
 $B ::= bBb | ba$

So, there are Left recursion and Left factoring, not suitable for Top-down predictive parser

$$A \rightarrow Ax | B$$

$$\downarrow$$

$$A \rightarrow BA'$$

$$A \rightarrow \alpha A' | \epsilon$$

$L ::= BbL' | aaaL'$
 $L' ::= baL' | \epsilon$
 Left recursion

$B ::= bB'$
 $B' ::= Bb | a$

$$A ::= \alpha \beta_1 | \alpha \beta_2$$

$$\downarrow$$

$$A = \alpha A'$$

$$A' \rightarrow \beta_1 | \beta_2$$

New grammar is

~~$L ::= BbL'$~~
 $L ::= aaaL' | BbL'$
 $L' ::= baL' | \epsilon$
 $B ::= bB'$
 $B' ::= Bb | a$

$A \rightarrow \alpha BB$
 $A \rightarrow \alpha B$

First

$First(L) = \{a, b\}$
 $First(L') = \{b, \epsilon\}$
 $First(B) = \{b\}$
 $First(B') = \{a, b\}$

Follow

$Follow(L) = \{\$ \}$
 $Follow(L') = \{\$ \}$
 $Follow(B) = \{b\}$
 $Follow(B') = \{b\}$

	a	b	\$
L	$L \rightarrow aaaL'$	$L \rightarrow BbL'$	
L'		$L' \rightarrow baL'$ $B \rightarrow Bbb$	$L' \rightarrow \epsilon$
B		$B \rightarrow bB'$	
B'	$B' \rightarrow a$	$B' \rightarrow Bb$	

$L \rightarrow aaaL'$

$L \rightarrow BbL'$

$L' \rightarrow baL'$

$L' \rightarrow \epsilon$

$B \rightarrow bB'$

$B' \rightarrow Bb$

$B' \rightarrow a$

4(b). $L \rightarrow Ra \mid Qba$
 $R \rightarrow aba \mid cabar \mid Rbc$
 $Q \rightarrow bbc \mid bc$

Left Recursion

$R \rightarrow abar' \mid cabar'$
 $R' \rightarrow bcR' \mid \epsilon$

Left Factoring

$Q \rightarrow bQ'$
 $Q' \rightarrow bc \mid c$

New Grammar:

$L \rightarrow Ra \mid Qba$
 $R \rightarrow abar' \mid cabar'$
 $R' \rightarrow bcR' \mid \epsilon$
 $Q \rightarrow bQ'$
 $Q' \rightarrow bc \mid c$

$A \rightarrow aBB$
 $A \rightarrow aB$

First

Follow

$First(L) = \{ a, c, b \}$

$Follow(L) = \{ \$ \}$

$First(R) = \{ a, c \}$

$Follow(R) = \{ a \}$

$First(R') = \{ b, \epsilon \}$

$Follow(R') = \{ a \}$

$First(Q) = \{ b \}$

$Follow(Q) = \{ b \}$

$First(Q') = \{ b, c \}$

$Follow(Q') = \{ b \}$

	a	b	c	\$
L	$L \rightarrow Ra$	$L \rightarrow Qba$	$L \rightarrow Ra$	
R	$R \rightarrow abar'$		$R \rightarrow cabar'$	
R'	$R' \rightarrow \epsilon$	$R' \rightarrow bcR'$		
Q		$Q \rightarrow bQ'$		
Q'		$Q' \rightarrow bc$	$Q' \rightarrow c$	

So LL(1)

Homework 2 Solution: CFG and Parsing

1. (10pts) Write a context-free grammar describing the syntax of variable declarations in the C language. In particular, each variable declaration should start with a type, followed by a sequence of variables and arrays being declared. The terminals of your grammar include

INT (the "int" keyword)
FLOAT (the "float" keyword)
ID (variable names)
IVAL (integer values)
FVAL (floating point values)
"[" "]" (left/right bracket)
"*" (the pointer operator)
"=" (the assignment operator)
";" (semicolon)
"," (comma)

Note that initialization is allowed inside variable declarations, but you do not need to ensure type correctness of the variable initializations (i.e., it is OK to initialize an integer variable with a floating point value). As a test case, your grammar should accept

```
int a = 5, b[100], c[100][200];  
float* pointer[3], *d = 0;
```

but should reject

```
int float a;  
float 34 a;
```

Solution: Context-free grammar for C variable declarations:

```
var_declarations ::= type var_declarators ;  
var_declarators ::= var_declarator | var_declarators, var_declarator  
var_declarator ::= var_declarator_id | var_declarator_id = var_initializer  
var_declarator_id ::= ID | * var_declarator_id | var_declarator_id [ IVAL ]  
var_initializer ::= IVAL | FVAL  
type ::= INT | FLOAT
```