Thomas Martin

Test-1

CMPSCI4250

16

MaxGradePoint:20

MaxTime:75Min

Note: Test is open book. Notes are not permitted. Each question worth is 4 grade points.

1. Convert this BNF to EBNF and show this grammar is ambiguous.

① <program> → <stmts> No Recursion

② <stmts> → <stmt> | <stmt> ; <stmts> ⇒ <stmt> { <stmt> }

③ <stmt> → <var> = <expr> no recursion

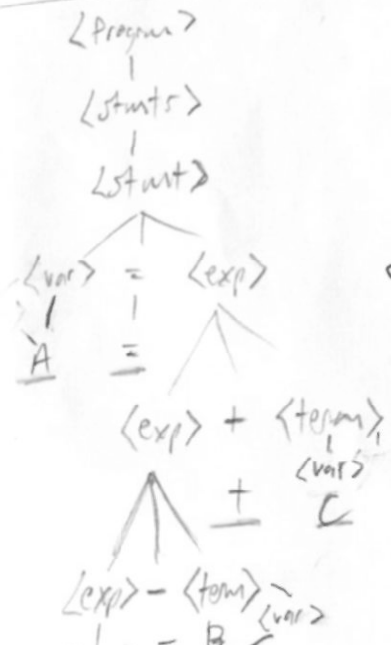
④ <var> → a | b | c | d No recursion

⑤ <expr> → <expr> + <term> | <expr> - <term> | <term> ⇒ ~~<expr> (+/-) <term> / <term>~~

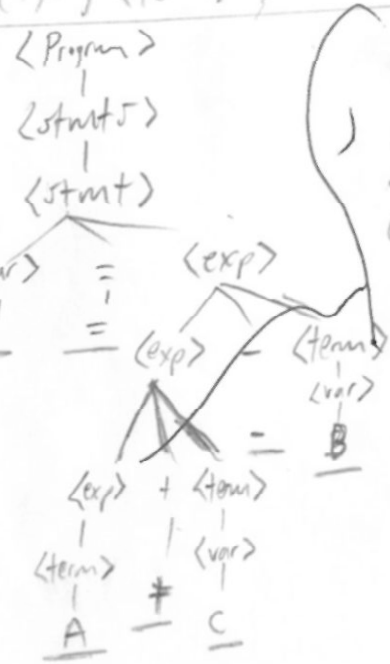
⑥ <term> → <var> | const | <expr> No recursion

⇒ ~~<term> { (+/-) <term> }~~ <term> { (+/-) <term> }

2.5



A = A - B + C



A = A + C - B

Sentence should remain same.

2 different parse trees for same sentence so Grammar is Ambiguous

Thomas Martin

First(Assign) = First(ID) = {A} First(Expr) = {id}

Follow(assign) = {$}

Follow(expr) = {+, *,), A}

First(ID) = {ε, ID}

thomas martin

So add first of producing symbol ie. first(expr)

<expr> followed by ε so include all producing symbols firsts.

2. Consider the following grammar:

<assign>=><id> = <expr> ε

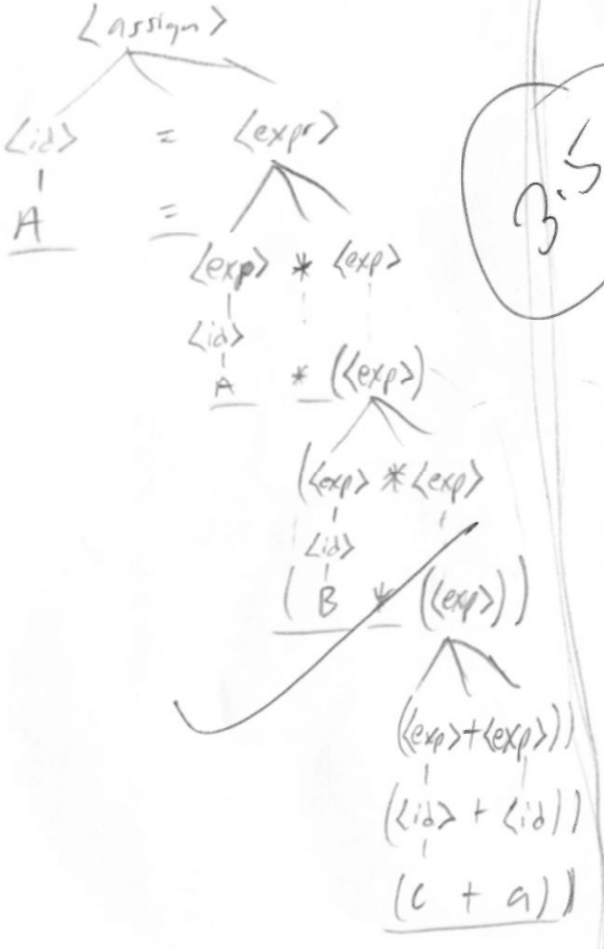
(R) parenthesis

<id> =>A | B | C

<expr>=><expr>+<expr>| <expr>*<expr>| (<expr>)| <id>

Show a parse tree for each of the following statements. Also, calculate first and follow for the grammar

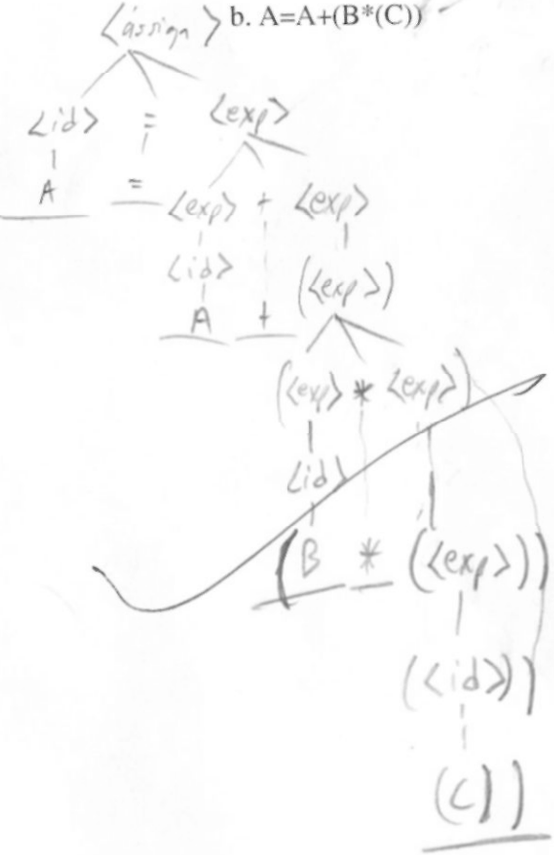
a. A=A*(B*(C+A))



A=A*(B*(C+A))

3.5

b. A=A+(B*(C))



A=A+(B*(C))

3. From the following grammar remove left recursion and for newly created grammar generate a parse tree for [redacted] id + id * id

i. E -> E + T / T

~~ii. E -> T~~

iii. T -> T * F / F

~~iv. T -> F~~

~~v. F -> (E)~~

vi. F -> id / (E)

E → E+T/T → E → +TE'
E' → TE'/ε

T → T*F/F → T → *FT'
T → T'*F

F → id/(E)
F → id/(E)

2

Thomas Martin #5

	Stack	Input	Action
	$	(id*id)*id$	S4
	$(4	id*id)*id$	S5
	$(4id5	*id)*id$	R6
4F=3	$(4F3	*id)*id$	R4
4T=2	$(4T2	*id)*id$	S7
	$(4T2*7	id)*id$	S5
	$(4T2*7id5	)*id$	R6
7F=10	$(4T2*7F10	)*id$	R3
4T=2	$(4T2	)*id$	R2
4E=8	$(4E8	)*id$	S11
	$(4E8)11	*id$	R5
$F=3	$F3	*id$	R4
$T=2	$T2	*id$	S7
	$T2*7	id$	S5
	$T2*7id5	$	R6
7F=10	$T2*7F10	$	R3
$T=2	$T2	$	R2
$E=1	$E1	$	ACCEPT

4

Remove Indirect & recursion

(2 I) $S \to Aa/Bb$ $A \to Aa/Abc/c/Sb$ $B \to bb$

$A \to Aa/Abc/c/Aab/Bbb$

$A \to A(a/bc/ab)/(\epsilon/Bbb)$

α β

Now...

$A \to cA'/BbbA'$

$A' \to aA'/BcA'/abA'/\epsilon$

Final Answer

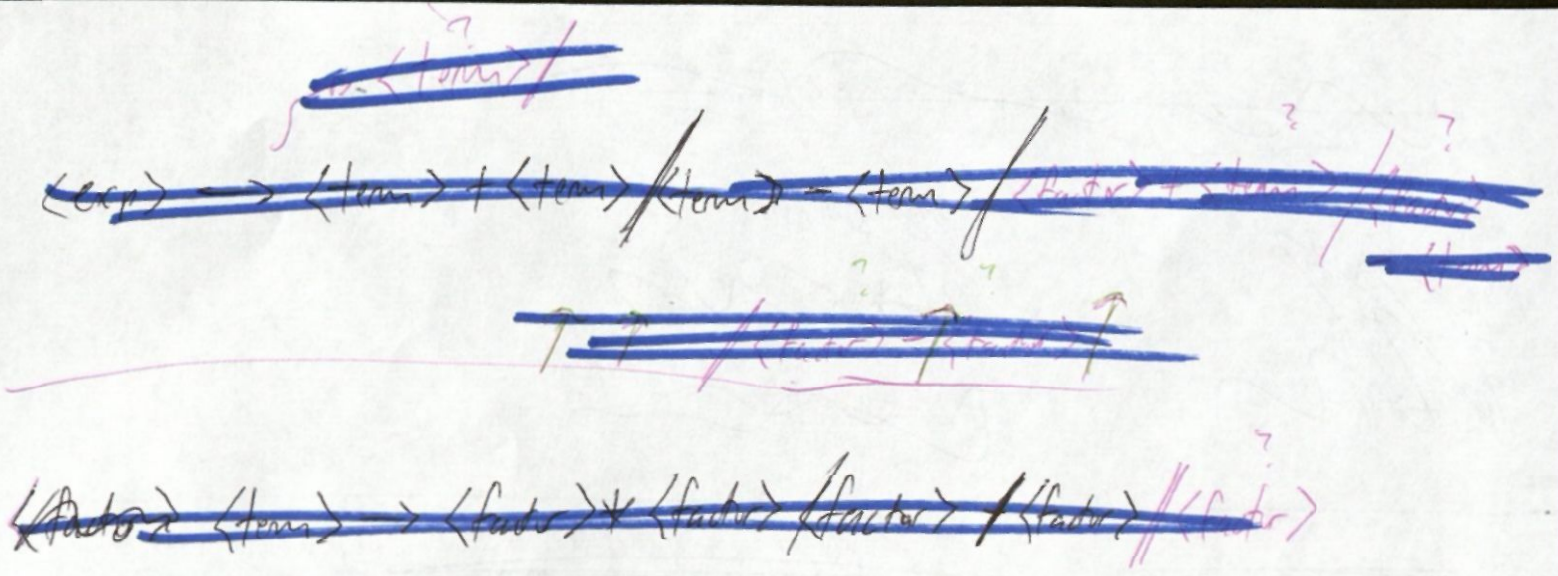
$S \to Aa/Bb$

$A \to cA'/BbbA'$

$A' \to aA'/BcA'/abA'/\epsilon$

$B \to bb$

(2 II)



Convert EBNF in to BNF

2 II

{exp} → <term> {(+|-) <term>}

<term> → <factor> {(*|/) <factor>}

{exp} → <exp> + <term> | {exp} - <term> | <term>

<term> → <term> * <factor> | <term> / <factor> | <factor>

Thomas Martin
Mishra CS4250
3/20/18 HW2

①

Production	Semantic Rules
$T \rightarrow FT'$	$T'.inh = F.val$
	$T.val = T'.syn$ (synthesized attribute)
$T' \rightarrow *FTI'$	$TI'.inh = T'.inh * F.val$
	$T'.syn = TI'.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow ID$	$F.val = id.lexval$

inh = inherited attribute
syn = synthesized attribute

- Synthesized attribute: taker value from child nodes. S-attributed.
- Inherited attribute: taker value from parent or sibling nodes. L-attributed

② Compute weakest precondition

a) `a = 2*b+1;` `b = a-3;` {b<0}

a-3<0
a<3

b) `a = 2*(b-1)-1` {a>0}

2*(b-1)-1>0
2b-2-1>0
2b>3
b>3/2

c) `if(a==b)` `b=2*a+1;` `else` `b=2*a;` {b>1}

NOT using parenthesis () as above

2*a+1>1 or 2
2a+1>1 2a+2>1
2a>0 2a>1/2
a>0 a>1/2

2*a>1
a>1/2
rule of consequence pg. 173

d) `if(x>y)` `y=2*x+1` `else` `y=3*x-1` {y>3}

2x+1>3
2x>2
x>1

3x-1>3
3x>4
x>4/3
rule of consequence pg. 173

(2)

if it is not a variable then actual type is intrinsic

Syntatic Rule: <assign> → <var> = <expr>

semantic Rule: <expr>.expectedtype ← <var>.actual_type

intrinsic is synthesized

inherited

intrinsic

- expected_type is ALWAYS inherited

Syntatic Rule: <var> = A/B/C

semantic Rule: "What can we calculate from this statement?"

<var>.actualtype ← Lookup string (variable)

Syntatic Rule: <expr> → <var> + <expr>

example A, B, C = int

a+b+c = <var> + <exp>

semantic Rule: iff (<var>.actual_type = <expr>.actualtype)

<expr>.actual_type = <var>.actual_type

else

<expr>.actual_type = real

predicate: <expr>.actualtype = <expr>.expected_type

{ 4th same as above but change + to * }

5th Syntatic Rule: <expr> = <var>

semantic: <expression>.actual_type = <var>.actual_type

Test1: CMPSCI-4250

Name Thomas Minton Date 4/11/18

Max Time:75 min Max Points:20

1. Write an attribute grammar for a language which impose following constraints on this grammar

<assign> →<var>=<expr> ⟹ S.R → expr.et = var.at

<var>→ A/B/C lookup (string)

<expr>-→<var>+<expr>/<var>*<expr>/<var>

a) Assignment statement will be valid only if the type of variable on left matches with the type of expression.

b) Type of variable can be int or float.

c) Expression can have different types of variables. If all variables are int then the type of expression will be int otherwise it will be real. Show intrinsic, inherited and synthesized attributes.

Show whether A=A+(B*C) will be a valid assignment or not. Type of A and C is int and type of B is Real. (5)

1) Syntax <assign> → <var> = <expr>

Semantics: <expr>.expected_type ← <var>.actual_type

2) Syntax <var> → A/B/C

Semantic: <var>.actual_type ← look-up(<var>.string)

3) Syntax: <expr> → <var> + <expr>

Semantics: <expr>.actualtype ← if(<var>.actual_type = int) and (<expr>.actual_type = int) then int, else real endif

Predicate: <expr>.actual_type == <expr>.expected_type

ON BACK

4) Syntax: <expr> → <var> * <expr>

Semantics: <expr>.actual_type ← if (<var>.actual_type = int) and
(<expr>.actual_type = int)
then int
else real
end if

Predicate: <expr>.actual_type == <expr>.expected_type

5) Syntax: <expr> → <var>

Semantics: <expr>.actual_type ⟸ <var>.actual_type

Predicate: <expr>.actual_type == <expr>.expected_type

int real int

A = A + (B * C) NOT valid b/c B is real

type A must match type of (R) side

✓ First we parse tree to check grammar. B/c ( ) parenthesis is missing from grammar, the statement is not valid.

var. ? →

expr.at → synthesized attribute

expr.et → inherited attribute

2. Compute the weakest precondition for the following sequence of assignment statements, for the post condition given. (4)

a) a := b/3 + 1;
b :=b – 3
{b = 0}

Correct

$b=0 \Rightarrow b=3-3=0$

$b=3 \Rightarrow \frac{3}{3}+1=a$

So $a=2$

b) a := 2*(b – 1) –1 given the post condition {a > 0)

$2(b-1)-1>0$

$2b-2-1>0$

$2b-3>0$

$2b>3$

$b>\frac{3}{2}$

c) if(a==b)
b=2*a+1
else
b=2*a-1
{b>1}

$2a+1>1$

$2a>0 \Rightarrow a>0$

$2a-1>1$

$2a>2$

$a>1$

d)
if(x>y)
y=2*x-1
else
y=3*x-1
{y>3}

$2x-1>3$

$2x>4$

$x>2$

weakest precondition

$3x-1>3$

$3x>4$

$x>4/3$

weakest precondition b/c satisfies both conditions

3. **Assume the following rules of associativity and precedence for expressions:**

Precedence *Highest* *, /, **not**

+, –, &, **mod**

– (unary)

=, /=, < , <=, >=, >

and

Lowest **or, xor**

Associativity *Left to right*

Show the order of evaluation of the following expressions by parenthesizing all subexpressions and placing a superscript on the right parenthesis to indicate order. For example, for the expression

a + b * c + d

the order of evaluation would be represented as (3)

$((a + (b * c)^1)^2 + d)^3$

a. *(a – b) / c & (d * e / a – 3)*

b. *a > b* ***xor*** *c* ***or*** *d <= 17*

c. *a * (b – 1) / c* ***mod*** *d*

d. *-a + b*

a) $(((a-b)^1/c)^3 \;\&\; (((d*e)^2/a)^4 - 3)^5)^6$ corresponds to & operator

b) $(((a>b)^1 \text{ XOR } c)^2 \text{ or } d<=17)^3$ $(((a>b)_1 \text{ XOR } (d<=17)_2)_3)_4$

c) $(((a*(b-1)^1)^2/c)^3 \bmod d)^4$ $a*(b-1)/c \bmod d$

d) $((-a)^1+b)^2$ 1

$(-(a+b)_1)_2$

5. Differentiate between row major and column major representation of array. State with example the difference between imperative and functional programming. (2)

Imperative uses a sequence of statements to determine how to reach a certain goal while functional programming was specifically created to support a pure functional approach.

Functional Languages → Python ; Scala

Imperative Languages → C++ ; Java

Row Major / Column Major format describe how 2-d arrays are mapped on 1-D of memory; Row Major array data is more likely to be in the cache (local) so row Major is more efficient.

2.5

6. Explain the meaning of Lambda expression. Write set of instructions to define a lamda expression for ax^3+bx^2+cx+d. (2)

A Lambda Expression specifies the parameters and mapping of a function. The lambda expression is the function itself, which is nameless. Untyped Lambda calculus is inspiration for functional programming languages.

(λ (a b c) (+ (+ (+ (* a (* x x x)) (* b (* x x))) (* c x)) d))

2

7. Explain the meaning of each line of this function. (1)

Write How CAR and CDR functions work in function programming Language.

```
DEFINE (member atm a_list)
(COND
        ((NULL? a_list) #F)
        ((EQ? atm (CAR a_lis)) #T)
        ((ELSE (member atm (CDR a_list)))
))
```

if a_list is empty return FALSE

if atom found in 1st element return TRUE

define atm as remaining elements of a_list that is not the first 2

- CAR returns 1st element of its list parameter
- CDR returns parameter list minus its first element

1

←----------END----------→

8 #1

11001 = 25

#4

$M_{bin}('01') = 0$

$M_{bin}('1') = 1$

$M_{bin}(<bin\_num>'0') = 2 * M_{bin}(<bin\_num>)$

$M_{bin}(<bin\_num>'1') = 2 * M_{bin}(<bin\_num>) + 1$

~~M('01') = 1100 + 2(1) + 1~~

[11001] = [1100] + 2(1) + 1 = [110] + 2(0) + 2(1) + 1 ☆

☆☆ [11] + 2(0) + 2(0) + 2(1) + 1 = [1] + 2(1) + 1

2

2.5