2/07/18

CS4250

- When deciding relationship... think what you CARE about. = 4610 advice

$A \rightarrow a[b]A$

$S \rightarrow A\{bA\}$

Answer: $A \rightarrow aA \mid abA$

$S \rightarrow SbA \mid A$

1) <program> → begin <stmt-list> end
2) <stmt-list> → <stmt> / <stmt> ; <stmt-list>
3) <stmt> → <var> = <expression>
~~<program> → <var> = <expression>~~
4) <expression> → <var> + <var> / <var> - <var> / <var>

Answer: 1) No recursion can not EBNF
2) <stmt-list> → <stmt> { ; <stmt> }
3) No recursion
4) No recursion // <expression> → <var> (+/-) <var> / <var> ~~A~~
~~A~~ — <var> [(+/-) <var>]
<stmt> ⇐ <var> = <var> [(+/-) var]

Page 4 on BACK

CS4250
2/7/18

Top Down Parser → ① recursion

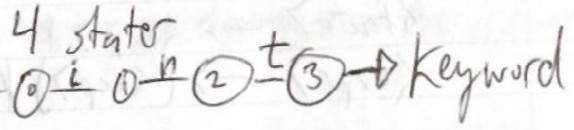
deterministic Non-deterministic

Page 3 is Page before this one!

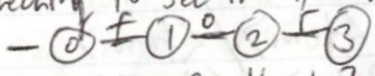
Finite Automata (DFA)/(NFA)

Keyword — int X, Y;

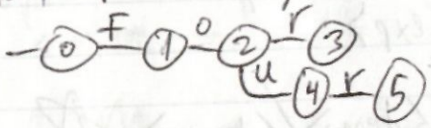
- int → keyword
  - ~~→ identifier~~
- if you want to redefine int you need



- Now checking to see if for is a keyword



- What about FOUR though?



→ We made another state.

- Just know purpose of Finite Automata?
  - → Used for checking

- Post Down Automata → Used for parsing
  When we did stack input next step table the stack was essentially Post Down Automata
- BNF → an extension is EBNF — the expressivity is the same
  - EBNF is shorthand for BNF; purpose is EBNF to reduce size of productions in BNF
- Suppose we have this grammar: $S \xrightarrow{BNF} aA \mid a$ → Non-recursive b/c No S on (R) side
  $S \to aA \mid a \to a[A]$ EBNF: $S \to a[A]$

Example – $B \to abS \mid ab$ Now change to EBNF $B \to ab[S]$

1st Rule – $IS \to$ if E then statement | iff Expression then Statement Else St.
Now in EBNF $IS \to$ IF E then St [Else St]

2nd Rule – Now $S \to aS \mid a$ → Yes recursive → {a, a(a), aa(a), aaa(a)...}
→ a{a} = a{ε, a, aa, aaa...}

3rd Rule – given $E \to E+T \mid E-T \mid E*T \mid E/T$ – only thing that differs is operator
$E \to E(+/-/*/\,/)T$

2/7/18

Page 2-

Example: <ident-list> → identifier | identifier, <ident-list>

EBNF: identifier [identifier]

this is language of this grammar

Example 2: <exp> → <exp> + <term> / <exp> - <term> / <term>

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

<factor> → <exp> * <factor> / <exp>

<expr> → (<exp>) / <ID>

<term>
<factor>
<term> / <factor>

1 <expr> → <expr> + <term> / <expr> - <term> / <term>

2 <term> → <term> * <factor> / <term> / <factor> / <factor>

3 <factor> → <exp> ** <factor> / <exp>

4 <exp> → (<expr>) / id

Answer: <expr> → <expr> (+, -) <term> / <term>

~~<term> → (+/-) <term>~~

Now rewrite 2) <term> → <factor> {(*/) factor}

Now rewrite 3) <factor> → <exp> {** <exp>}

4) No change

PAGE 3 is Previous Page

2/12/18 Page 4
CS 4250

- Continuing example w/ grammar on previous page...

<exp> actual type <exp> ≠ <var>

A = B + C + D is NOT valid b/c A's type is int & B is float.

Defining Attribute grammar: Syntax Rule & Semantic Rule.

Expr <assign> → <var> = <exp>

<exp> → <var> + <var> / <var>

<var> → A / B / C inherited

in Book — Rules for this grammar

1) <assign> → <var> = <exp> → Syntax Rule

Semantic Rule → <exp> expected type = <var> actual type

synthesized

2) <exp> → <var> + <var> → Syntax Rule

Semantic Rule ⟹ iff (<var>1 actual type = int) and
iff (<var>2 actual type = int) then
<expr> · actual type = int;
else
<expr> · actual type = float;

3) <exp> → <var> → Syntax Rule

Semantic Rule <exp> actual type = <var> actual type

4) <var> → A / B / C

Semantic Rule lookup string A;

~~[illegible]~~

▷ Add 1 more rule to 2nd syntax rule

<exp> · actual type = <exp> · expected type

p. 159 question 19

- Write Att. Grammar for same example as before but w/ newer rules

Rules: ① Data types can not be mixed in expression

② Data types cannot be mixed in exp but assignment statements need not be same

1) <assign> → <var> = <exp> Does NOT need new rule

2) <exp> → <var> + <var>

semantic Rule: iff (<var>1.actualtype ≠ <var>2.actualtype)

then print ("Expression is NOT valid")

else

<exp>.actualtype = <var>.type

3) <exp> → <var> Does NOT need new rule

4) All good w/ this one...?

New Example

<assign> → <id> = <exp>

<id> → A/B/C

<exp> → <id> + <exp> / <id> * <exp> / <exp> / <id>

Same Rules as are Above. Write grammar. w/ id type = exp type

1) <assign> → <id> = <exp>

<exp>.expectedtype = <id> actualtype

2 <id> → A/B/C

3) <exp> → <id> + <exp>

<exp>.actualtype

iff (<id> actualtype = <exp>1.actualtype)

then (<exp>.actualtype = <id>.actualtype)

else (<exp>.actualtype = float)

$a^n b^n c^n$ → Can you write CFG for this?

$a^n b^n$ → Can you write CFG for this?

↳ #A = #B so define recursively

S → a S b / ∈

↳ Sentences of this { ∈, ab, aabb, aaabbb ... }

~~bad~~

S → abS / ∈

↳ Sentences of this { ∈, ab, abab, ababab ... }

Continued on Back

2/14/18
CS4250

Example ☐ Find First & Follow ~~S ABCDE~~

- First ← Can start w/ any non-terminal symbol
- First(S) = First(A) = {a, ε} + {b, ε}
  - now check A (terminal symbols)
  - ↳ ~~ε~~
  - First(S) = First(A) + First(B) + First(C)
    - Now check B

S → ~~A~~BCDE (ε ε)
A → a/ε
B → b/ε
C → c
D → d/ε
E → e/ε

- Follow(S) = {$}
  - is there any production w/ S on (R) side? No so $ only symbol
  - S → ABCDE
- Follow(A) → First of B (B/ε) → First(C) = (c ~~ε~~)
  - Follow(A) = {b, c}
- Follow(B) = {c} → Follow of B or First of C
- Follow(C) = {d, e, $}
  - "C gets followed by ε so everything that follows S follows C"
- Follow(d) = {e, $}
- Follow(e) = {$}

There WILL be a First & Follow question on Exam

Example 2 ☐ S → Bb/cd
B → aB/ε
C → cC/ε

- First(S) = ~~{a, b, c, d, ε}~~ → ~~No production w/ S on (R) side~~ First(B) + First(C)
- ~~First(B)~~ First(S) = b (replace B w/ ε) {a, c, d, ε}
- First(B) = {a, ε}
- First(C) = {c, ε}
- Follow(S) = {$} → No S on (R) side of productions
- Follow(B) = {b}
- Follow(C) = {d}

- S → (S)/ε (L parenthesis)
- First(S) = {(, ε}
- Follow(S) = {$,)} (R parenthesis)

- How many phases are there in compiler? What is read/writeability? What is purpose of Lexical Analyzer.

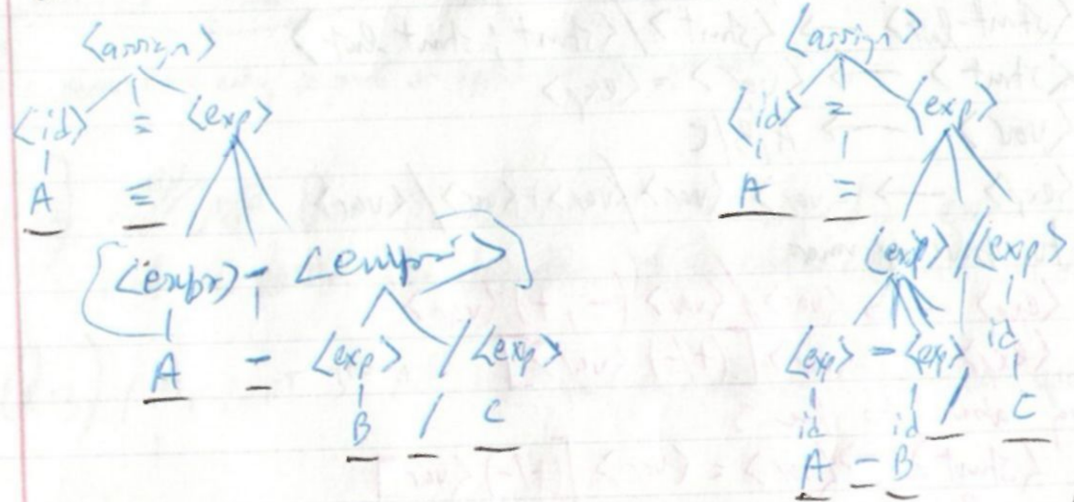
★ ALL Questions are from Textbook ★ 5 questions

2/19/18
CS4250

Given grammar <assign> → <id> = <exp>   Also, is this gr
<id> → A / B / C
<exp> → <exp> - <exp> / <exp> / <exp> / id

Can you draw a parse tree that will show this grammar A - B / C - A → NO!
" " " " " " " " " " " A = A - B / C

Answer:



- Now write EBNF for above example:

Answer: <assign> → [id, exp]

<exp> → ~~<expr>~~ <id> / { <exp> (

Rules for EBNF (symbol...)
1. [ ] appears 0 (zero) or 1 (once)
2. { } (Recursive Bracket) symbol appears 1 (one) to ∞ (infinity) times
3. ( ) to combine operator

is it recursive?
E → E+T / E/T Yes! E is producing E → EBNF for this is E → T {+T}
S → A+A / A No!

2/19/18

$E \to E+T / E-T / T$ Find EBNF

Answer: $E \to E(+/-)T / T$

$E \to T \{(+/-)T\}$

Now... $A \to T / TE$ → NOT Recursive

→ EBNF = $A \to T[E]$

FOR EBNF Apply the rules!

Grammer for writing small language

1) <program> → begin <stmt-list> end
2) <stmt-list> → <stmt> / <stmt ; stmt list>
3) <stmt> → <var> = <exp>
4) <var> → A/B/C
5) <exp> → <var> - <var> / <var> + <var> / <var>

Apply EBNF to above grammer

Answer: <exp> → <var> / <var> (-, +) <var>

<exp> → <var> [(+/-) <var>]

Now plug above into line 3

<stmt> → <var> = <var> [(+/-) <var]

~~Now combine line 3 & line 2~~ line 2 is recursive so can reduce it

<stmt-list> → {<stmt> ; } <stmt>

Chapter 4 → check grammar LL1 for Left Recursion

$S \to Aa / Bb$
$A \to Aa / Abc / c / Sb$
$B \to bb$

This grammar has indirect (L) recursion
The S can produce indirect (L) recursion

$A \to Aa / Abc / c / (Aa / Bb)b$

$A(a/bc/ab)$ Aab/Bbb

$A(a/bc/ab) / (c/Bbb)$

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

Now Expand it: $S \to Aa / Bb$

$A \to cA' / BbbA'$
$A' \to aA' / bcA' / abA' / \epsilon$
$B \to bb$

2/26/18
CS 4250

Attribute grammar → (CFG) → (T, N, S, P, A)

are attributes? → attributes are related to Non-Terminals

Must write 2 rules → ① Syntax Rule & ② Semantics Rule

Before we are writing program language where assignment will be valid if the type of variable matches type of expression

CHAPTER 3

1) Assignment → var = exp

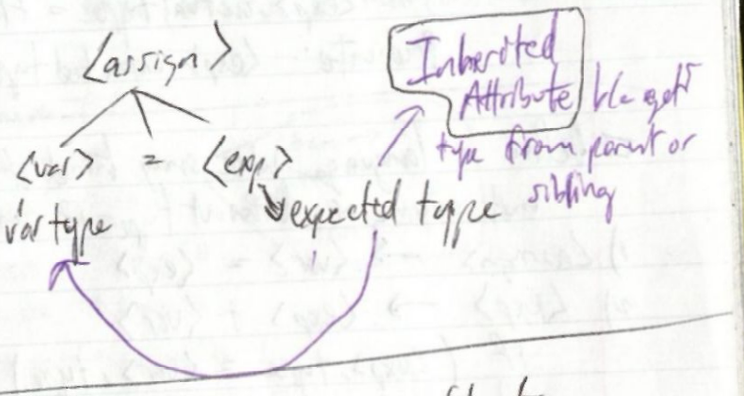
<assign> → <var> = <exp>

<exp> → <var> + <var> / <var>

<var> → A / B / C

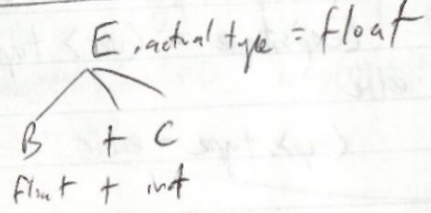
A	B	C
int	float	int

type is an attribute of <var>



We want to match ~~exp.expectedtype~~ exp.expected type = exp.actual type

Synthesized Attributes are those attributes which get their attributes from their children or from the program



Intrinsic Attribute (a type of synthesized attribute)

1) <assign> → <var> = <exp>

Semantic Rule: exp.expected type = var.actual type /// this is inherited attribute specifically Intrinsic

2) <exp> → <var> + <var> /// var → int. or var → float

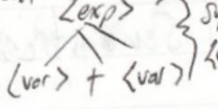
Semantic Rule: if (<var 1>.type == <var 2>.type)

<exp>.actual type = <var 1>.type

else

<exp> actual type = float

Predicate exp.actual type = exp.expected type



3) <exp> → <var>

Semantic Rule <exp>.actual type = <var>.actual type

Predicate: exp.actual type = exp.expected type

2/26/18
6:54250

New Grammar: <assign> → <var> = <exp>
<exp> → <exp> + <var> / <var>
<var> → A/B/C

"Attribute grammar is best when semantic Rules are minimized"

1) <assign> → <var> = <exp>
Semantic Rule: <exp>.expected type = <var>.actual type
2) <exp> → <exp> + <var>
Semantic Rule: if (<exp>.actual type == <var>.actual type)
<exp>.actual type = <var>.actual type
else
<exp>.actual type = float
Predicate: <exp>.expected type = <exp>.actual type ? ?

□ Define a language where any kind of assignment operator is allowed but mixing of different types of variables is NOT allowed.

1) <assign> → <var> = <exp>
2) <exp> → <exp> + <var>
if (<exp>.type = <var>.type)
<exp>.type = <var>.type
else
<exp>.type = error

- semantic information

Binary	Decimal
00	0
01	1
10	2
11	3

Mathematical = Recursive

bin → bin '1'
bin → '0'
bin → bin '1'
bin → bin '0'

- Denotational Semantics

bin → '1'
M_b ('1') = 1
M_b ('0') = 0
M_b (bin '0') = 2 * M_b (bin) + 0
M_b (bin '1') = 2 * M_b (bin) + 1

1111 = 2x(111) + 1 = 2·(2·(11) + 1) + 1 = 2·(2·(2

111 = (11) 1 = (2x3) + 1
110 = (11) 0 = 2x3 + 0

2/28/18
CS4250

How to check the program? → Use Semantics

- Denotational Semantics powerpoint slide → Chapter 3
- Writing a program to convert binary # to decimal #... How do you know if it's correct?
  Apply Denotational Semantics to write a mathematical function that can convert any binary # to decimal #. Math F(x) → F(x) that has to be recursive. A language based on math F(x): Lambda Calculus.

bin → (0)
bin → (1)
bin → bin (1)
bin → bin (0)
} Create any binary # w/ this

decimal#
123 → (12)x10 +3

binary
$101_2 = (10_2) \times 2 + 1$

Mathematical Function → $M_{bin}('0') = 0$
$M_{bin}('1') = 1$
$M_{bin}(bin('1')) = 2 \cdot M_{bin}(bin) + 1$
$M_{bin}(bin('0')) = 2 \cdot M_{bin}(bin) + 0$

- Any # in base B will have B digits

{0;1... 77 734

base → '0'
base → '1'
base → 'b-1'
base → base '0'
base → base '1'
base → base 'b-1'

base → '2'
base → '3'
...
base → 'b-1'

base
base '4'
base '3'
base
'7'

$M_b('0') = 0$
$M_b('1') = 1$
$M_b('2') = 2$
skip some digits & $M_b('b-1') = b-1$

$M_b(base '0') = b \cdot M_b(base) + 0$
$M_b(base '1') = b \cdot M_b(base) + 1$
$M_b(base 'b-1') = b \cdot M_b(base) + (b-1)$

||| → '11' 1
$b \cdot p(11) + 1$

LISP - Functional Programming → write everything in mathematical f(x)'s.
- Diff between f(x) & imperative programming: $x = y = (+) = x+y$ vs $1\ 2\ (+) = 3$
  program → var → state so program changes variables states so we have to write mathematical f(x)'s that show states.

Chapter 3 example PAGE 164 example under 3.5.2.3 header
- Writing a denotational expression for a mathematical expression

⋮

- Axiomatic Semantics → Reverse Engineering

3/19/18
CS 4250

Type Binding

Static binding → global variables mostly

- Local variable in any f(x)? → use stack
- pointer? → use heap

LET (e (+ (+ x y) 3) x+y+3)

Chapter 5

a+b/c-d → top/bottom

(LET ((top (+ a b))
(bottom (- c d)))
(/ top bottom))

a+b-3 → top
c+d-4 → bottom
(name for each)

→ (Let ((top (- (+ a b) 3))
(bottom (- (+ c d) 4)))
(/ top bottom))

- ML → a functional programming language → No parathesis in ML

Let
val name1 = expression1
val namen = expressionn
in expression
end;

Let
val top = a + b
val bottom = c - d
in top / bottom
end;

Chapter 6

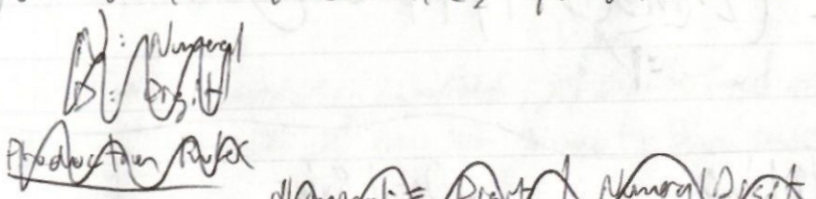
Advantage of primitive type?? Can define something in any language your self. ALREADY DEFINED OPERATORS

- if you are worried your data is critical & you want that data private you must use Object Oriented Programming

3/21/18
CS4250

1) What is attribute grammar?
▷ an extension of CFG where we add an symbol called Attribute.

2) Write denotational semantics for octal numbers?



Binary numbers —meaning→ Decimal value
Octal # —meaning→ Decimal value

$B \rightarrow '0'$	$M_B('0') = 0$
$B \rightarrow '1'$	$M_B('1') = 1$
$B \rightarrow B'0'$	$M_B(B'0') = 2 \times M_B(B) + 0$
$B \rightarrow B'1'$	$M_B(B'1') = 2 \times M_B(B) + 1$

For octal:

$O \rightarrow '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'$

$O \rightarrow O\ 'd'$ $\quad 0 \le d \le 7$

$M_O('d') \rightarrow d$

$M_O('0') \rightarrow 0 \quad M_O('1') \rightarrow 1 \ldots M_O('7') \rightarrow 7$

$M_O(O'd') \rightarrow 8 \times M_O(O) + d$

$M_B('0') = 0$
$M_B('1') = 1$
$M_B(B'0') = 2 \cdot M_B(B) + 0$
$M_B(B'1') = 2 \cdot M_B(B) + 1$

"Denotational Semantics"

3/21/18
CS 4250

$M_B(1101) \rightarrow M_B(110'1')$

$\rightarrow 2 \cdot M_B(110) \rightarrow 2 \cdot (2 \cdot M_B(11) + 0) + 1$ ★

★ $= 2^2 M_0(11) + 1 = 2^2 (2 \cdot \underbrace{M_B(1)}_{=1} + 1) + 1 =$ ☺

☺ $= 2^3 \cdot 1 + 2^2 \cdot 1$

Calculate decimal value of $(421)_8$:

START: $M_0(42'1') \rightarrow 8 \cdot M_0(4'2') + 1$ ★

★ $= 8 \cdot (8 \cdot M_0(4) + 2) + 1$ ☺

☺ ~~$= 8^2 \cdot 4 + 2 \cdot$~~ ~~$= 2065$~~

$= 8^2 \cdot M_0(4) + 8 \cdot 2 + 1$ ✡

✡ ~~$= 8^2 \cdot 4 + 8 \cdot 2 + 1$~~ $= 8^2 \cdot 4 + 8 \cdot 2 + 1$

Octal Rules

$M_0('0') = 0$

$M_0('1') = 1 \ldots M_0('7') = 7$

$M_0('0'0') = 8 \cdot M_0('0') + 0$

$M_0('0'7') = 8 \cdot M_0('0') + 7$

$M_B(1011)_8$?, ?

$M_0(1011) \rightarrow M_B(101'1') \rightarrow 8 \cdot M_0(1'01') + 1 \rightarrow$

$8 \cdot (8 \cdot M_B(10) + 1) + 1 \rightarrow 8^2 M_B(10) + 2 \rightarrow 8^2 (8 \cdot \underbrace{M_B(1)}_{=1} + 0) + 1$

64.9

Lambda grammar

- Use denotational method to find meaning of # prog
  denotational semantics

Attribute grammar

<assign> → <var> = <exp>

<exp> → <var> + <var> / <var>

<var> → A / B / C

i) exp type → var type

- What will be intrinsic, inherited, ~~&~~ synthesized attributes in this grammar
- Specify attribute ~~&~~ show how to specify these conditions in grammar?

A ~~=~~ B + C

A → Real / float
B → Real / Float
C → int

{ expected type → 'exp'
  actual type → 'exp' }

<assign> → <var> = <exp>
  exp.expectedtype = var.actualtype
<exp> → <var> + <var>
  if (var1.actualtype = var2.actualtype)
    exp.actualtype = var.actualtype
  else
    exp.actualtype = Real / float

A = C + D
PREDICATE → Check in textbook!!!!!

@ least 1 textbook question on Exam 2

"Thru is grammar, identify which will be synthesized attribute?"

"This is grammar... identify which will be inherited attribute?"

Chapter 7

① variable
② data structure
③ Expression
} 3 parts of a program

4/2/18
CS4250

What is evaluation for operator for a+b*c+d?

→ a+(b*c)$_1$+d ← this signifies order of operation

→ (a+(b*c)$_1$)$_2$+d

→ ((a+(b*c)$_1$)$_2$+d)$_3$

- Highest level → *, /, NOT

Associativity: Left to Right

+, -, &, mod

- (unary)

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

Lowest level → OR, NOR

(1) a*b-1+c

→ (a*b)$_1$ -1 +c

→ ((a*b)$_1$ -1)$_2$ +c

→ (((a*b)$_1$ -1)$_2$ +c)$_3$

(2) a*(b-1)/c mod d

~~→ a*(b-1)/c mod d~~

→ a*(b-1)$_1$ / c mod d

↔ (a*(b-1)$_1$)$_2$

→ ((a*(b-1)$_1$)$_2$ / c)$_3$ mod d

↔ (((a*(b-1)$_1$)$_2$ / c)$_3$ mod d)$_4$

4/2/18
CS4250
page2

TEST 2 Question like this. No wacky operator stuff.

example a > b XOR c or d <= 17

→ (a > b)1 XOR c or d <= 17

→ (a > b)1 XOR c or (d <= 17)2

→ ((a > b)1 XOR c)3 OR (d <= 17)2)4

---

① (a - b) / c ∧ (d * e / a - 3)

→ (a - b)1 / c ∧ (d * e / a - 3)

→ (a - b)1 / c ∧ (((d * e)2 / a)3 - 3)4

((a - b)1 / c)5 ∧ (((d * e)2 / a)3 - 3)4

CAR ∧ CDR definitely ON exam.

- What complex types can be defined in what languages??

- No Chapter 4
What kind of complex data types can be be defined in what languages??
implicit / explicit Show examples

4/23/18
CS4250

Ch. 9 ① Process Abstraction → just focus on process / procedural oriented language
C / C++

OR

② Data Abstraction → More weightage to data

→ Way to check if modular. To test big program, divide it into smaller modules.

→ provide more weightage to data. Provide more SECURITY to data
by ~~accept~~ encapsulating member functions.

Data + Member Function = CLASS

polymorphism → f(x) ~~overload~~ overloading → f(x)'s same but diff in # or type of parameter

→ operator overloading

~~Chapter~~ Ch. 10

4/25/18
CS 4250

IP
① Instr. Fetch
② Instruct. Record?
③ Operator Fetch
④ Execution
} Instruction Pipelining

I_1, I_2, I_3 | IF, ID, FO, FI = 4 clock cycles. I_2 is on FO stage when I_1 is on FI. So I_2 & I_3 only add 1 clock cycle each.

Total time = 4 + 1 + 1 = 6 clock cycles

Total time = $k + (n-1)$
K = number of stages
n = number of instruction

} Important for Operating System

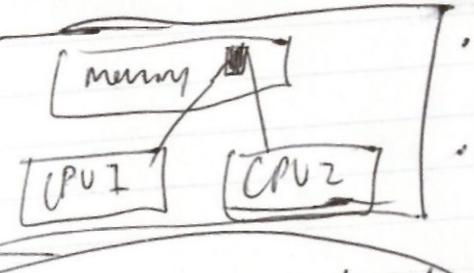
• Pipelining WILL decrease execution time.

Pipe Lining
• Have to write parallel processing to implement pipelining

- Heavy weight Tasks → Process
- Lightweight Task → Thread

Architecture ① Shared Memory Architecture
② Passing Messages Architecture

computer →

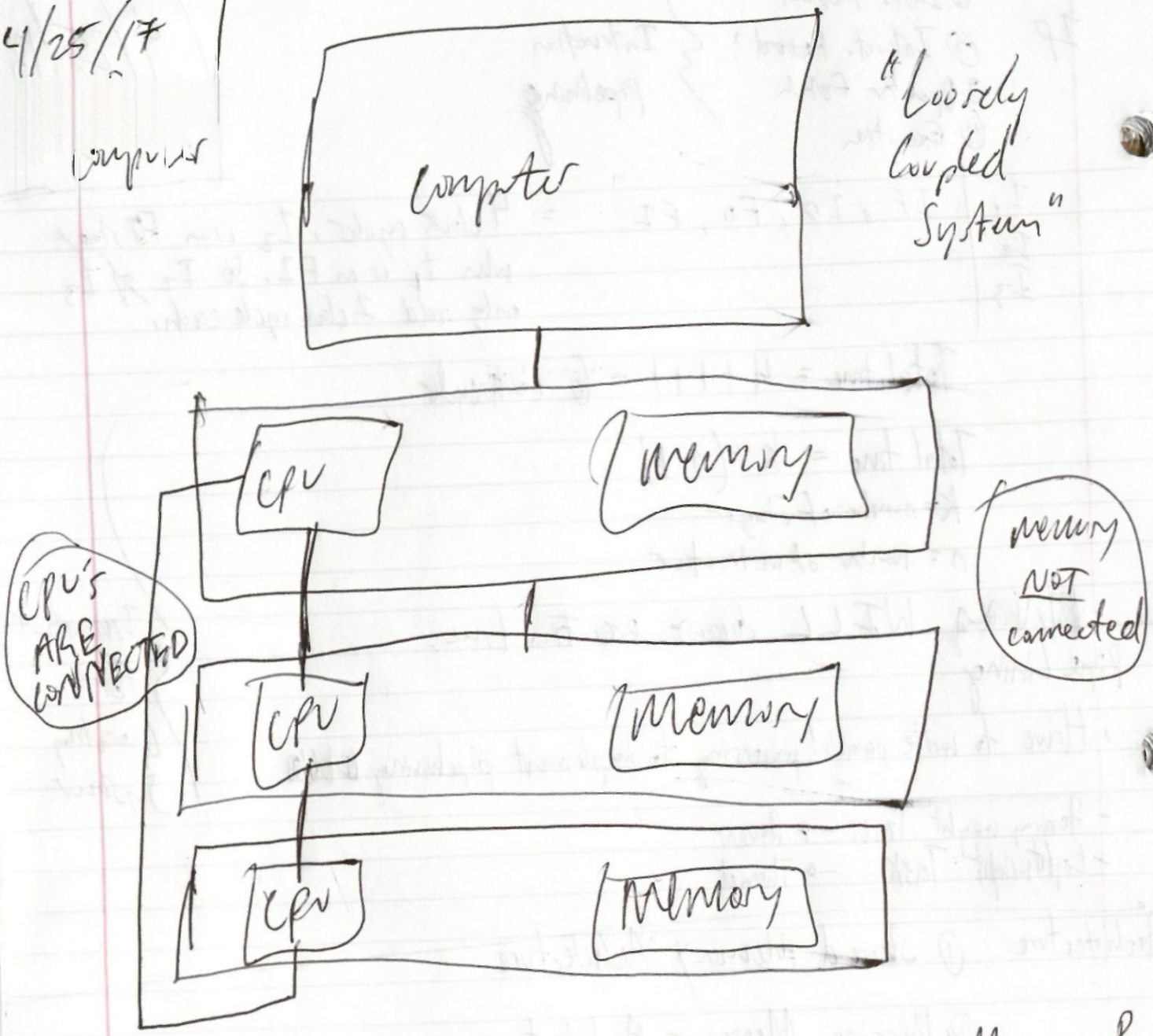


• 2 diff processors can access same memory locations @ same time.
• Need Semaphore.

"Shared Architecture"

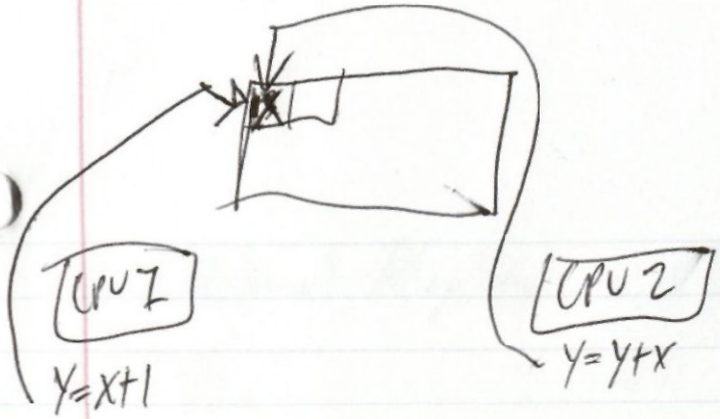
4/25/17

computer



* To implement parallel programing we need to implement 'Message Passing

4/25/18
CS4250



"Explain Semaphores" use this example.

- Both instructions running @ same time.
- These instructions are Dependent
  - Must implement Semaphores !! Use Semaphore provider
  ~~mark~~ variable to protect memory

wait → signal → retrieve

FINAL

Ch. 1; 3; 4; 5; 6; 7; 9; 10

Ch. 16; 15

Ch 11 → ~~How to~~ How implement synchron?

Answer: Semaphore, ~~etc~~ etc.

Ch 12

- Passing parameters
- Call by value
- Parameter passing
- Slide 23 ch. 9
- Activation Record Method Ch 10.

Slide 12 → 22 ch. 10

5/2/18
CS4250

- Final very similar to Test 1 ; Test 2;
- ~~4~~ Homework assignments
- Chapter 3 & 4 → 35% of final

LR Parsing Table

1) $E \to E+T$
2) $E \to T$
3) $T \to T*F$
4) $T \to F$
5) $F \to (E)$
6) $F \to id$

id + id * id (Bottom-Up Parser)

~~STACK~~	Input	Action
0	id + id * id $	S5
0 id 5	+ id * id $	R6
0 F 3	+ id * id $	R4
0 T 2	+ id * id $	R2
0 E 1	+ id * id $	S6
0 E 1 + 6	id * id $	S5
0 E 1 + 6 id 5	* id $	R6
0 E 1 + 6 F 3	* id $	R4
0 E 1 + 6 T 9	* id $	S7
0 E 1 + 6 T 9 * 7	id $	S5
0 E 1 + 6 T 9 * 7 id 5	$	R6
0 E 1 + 6 T 9 * 7 F 10	$	R3
0 E 1 + 6 T 9	$	R1

0 id = S5
5 + = R6
3 + = R4
1 + = S6
5 * = R6

0F gives state 3
0T gives state 2
6F gives state 3

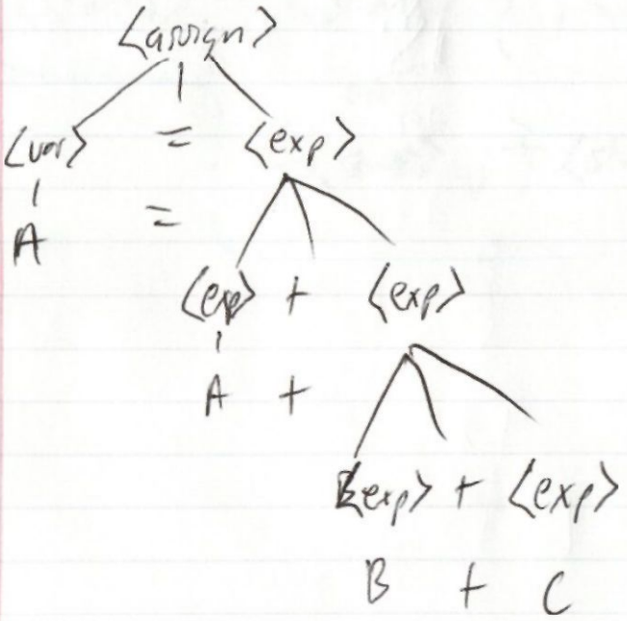
only difference is operand so ambigeous

⟨assign⟩ → ⟨var⟩ = ⟨exp⟩

⟨exp⟩ → ⟨exp⟩ + ⟨exp⟩ / ⟨exp⟩ * ⟨exp⟩ / ⟨var⟩

⟨var⟩ → A/B/C/D

- Show this grammar is ambigeous
- Draw a parse tree for A+B+C → NOT POSSIBLE!!!
  MUST start w/ ⟨assign⟩ → ⟨var⟩ = ⟨exp⟩ but A+B+C has no (=)
- Draw parse tree for A = A+B+C



Slide 1-11 "An example grammar"

- Convert EBNF → BNF

<program> → <stmts>

<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <exp>

<var> → a/b/c/d

<exp> → <term> + <term> / <term> - <term>

<term> →

- To reduce length of a production use { } OR [ ]

<exp> → <term> (+/-) <term>

<stmts> → <stmt> { , <stmt> }

OR

{ <stmt> ; } <stmt>

- Decimal number
  Bin number
} save from last 2

If (B=0) → we don't want to execute this statement

  B = B+1;

else

  B = B-1;

end

post condition B > 1

B = B+1 or B = B-1

post: B>1 post: B>1

B>0 B>2

B>0

B = B+1

B>1

B>0

B = B-1

B>-1

Calculate weakest precondition for:

```
if (x<y)
    x = x+1
else
    x = 3*x
post condition x<0
```

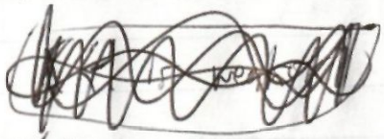
Start: Check `x = x+1` ~~&~~ `x = 3*x` statements

`x = x+1`

post: $x<0$

$x+1<1 \Rightarrow x<-1$

$x<0$ gives output $x<1$



$x<-1$ is weakest precondition

`x = 3*x`

post $x<0$

$3*x<0 \Rightarrow x<0$

gives output $x<0$

Ch. 15 & 16 prolog & Lisp

25% Final prolog
15% f(x) programming (Lisp)
10% ch 13 concurrency
35% Ch. 3 & 4
5% Ch. 9 & 10
5% Ch 1 → what is readability/writeability

- Programming Language needs to be suited for the process !!!

- John likes everything.
  likes (John, Everything);

- Men are Awesome
  ∀x:man Awesome (X);

∀x:man = "For all x in man"

- Nobody is perfect
  ¬∀x:man perfect (x);

¬∀x:man perfect (x);

~~[illegible]~~

- If Fred is the Father of Mike
  then Fred is ancestor of Mike

  ancestor (Fred, Mike) ⊂ Father (Fred, Mike)