COL728 Midterm Exam

Viraj Agashe

TOTAL POINTS

29 / 40

QUESTION 1

1 Is ambiguous grammar LR(1) 2 / 2

✓ **+ 1 pts** **Correctly identified that the grammar is not LR(1)**

✓ **+ 1 pts** **Correct argument for the above claim**

+ 0 pts Incorrect / Unattempted

QUESTION 2

2 Predictive parsing based on length 3 / 4

✓ **+ 1 pts** **Correctly identified that the parsing scheme is incorrect**

✓ **+ 3 pts** **Correct explanation of the above claim**

+ 0 pts Incorrect / Unattempted

- 1 Point adjustment

Lacking key argument : non-terminating application of rule E -> E + T

QUESTION 3

3 First, Follow, LR0 7 / 10

✓ **+ 1 pts** **$${\tt First}(S) = \{c, d\}$$**

✓ **+ 1 pts** **$${\tt First}(E) = \{c, d\}$$**

✓ **+ 1 pts** **$${\tt Follow}(S) = \{\}$$**

✓ **+ 1 pts** **$${\tt Follow}(E) = \{a,b,c,d,\$\}$$**

+ 6 pts Correct 3c: Grammar is not LR(0) due to S-R conflict / Ambiguity

✓ **+ 0 pts** **Reasoning for 3c not correct/unclear**

+ 0 pts Incorrect / Not attempted

+ 3 Point adjustment

Your automaton has incorrect states. For instance: after taking c, E and then taking E, you would have other items like E -> E.Ea and so on.

QUESTION 4

4 LL(k) vs LR(k) : advantages and disadvantages 4 / 4

✓ **- 0 pts** **Correct**

- 2 pts Incorrect advantage of $${\tt LL}(k)$$ over $${\tt LR}(k)$$

- 2 pts Incorrect disadvantage of $${\tt LL}(k)$$ over $${\tt LR}(k)$$

- 4 pts Not Attempted

QUESTION 5

5 Lexing error handling 4 / 4

✓ **- 0 pts** **Correct**

- 2 pts Unclear/incomplete explanation

- 3 pts Error token class not mentioned/explained

- 4 pts Incorrect/Unattempted

QUESTION 6

6 Typing rules : soundness and permissiveness 4 / 6

+ 3 pts Part (a) Correct

✓ **+ 1 pts** **Part (a) unclear explanation**

+ 0 pts Part (a) are Incorrect/Unattempted

✓ **+ 3 pts** **Part (b) Correct**

+ 1 pts Part (b) unclear explanation

+ 0 pts Part (b) are Incorrect/Unattempted

QUESTION 7

7 Dynamic scoping implementation 2 / 5

- 0 pts Correct

✓ **- 3 pts** **Partially correct.**

- 5 pts Incorrect

- 5 pts Not attempted/incomplete

A single, global table of bindings won't suffice -- you would need a stack of scopes.

QUESTION 8

8 Code Generation where Caller does most work 3 / 5

- **0 pts** Correct

Incorrect

- **2.5 pts** set/reset fp
- **2.5 pts** args unwinding

- **5 pts** Incomplete/Not attempted

- 2 Point adjustment

`ra` is set by `jal` insn. You cannot delegate its setup to caller.

VIRAJ AGASHE
2020CS10567

COL728 Midterm Exam
Compiler Design
Sem I, 2022-23

Answer all eight questions — Max. Marks: 40

1. Consider the ambiguous grammar shown below:

   $E \rightarrow E - E \mid 1 \mid 2 \mid 3$

   The "-" operator is not associative, and thus this grammar is ambiguous. Is this grammar LR(1)? Briefly explain your answer. [2]

Ans. The reductions $E \rightarrow 1_\Delta$, $E \rightarrow 2_\Delta$, $E \rightarrow 3_\Delta$ will be unambiguous, as they cannot appear in any other context (no SR conflicts), and can reduce to only one nonterm (no R-R conflicts).

~~Further, there is no reduction~~
~~E / E . - E~~
Further, when we are at $E \rightarrow E - E_\Delta$ there is no other item in its closure

~~Since there are no other productions for E,~~
Consider the state,

$E \rightarrow E - {}_\Delta E$
$E \rightarrow {}_\Delta E - E$

$\xrightarrow{E}$

$E \rightarrow E - E_\Delta$
$E \rightarrow E_\Delta - E$

Even if we see the lookahead token, which is −, note that even if we reduce, − can still be the next token. Eg.

Stack $E - E . - 3$ So, there is S-R conflict & not LR(0).

2. Consider the following grammar:
   $E \rightarrow E + T \mid T$
   $T \rightarrow 1 \mid 2 \mid 3$

   Professor X suggests a predictive parsing scheme where depending on the number of remaining tokens in the string, the parser decides which production to use. For example, if the number of remaining tokens in the string is only 1 (excluding "$"), then the parser will use the production "E→ T"; else it would use the production "E→E + T". Do you agree with Professor X? Briefly explain the reason for your agreement/disagreement. [4]

Ans. ~~Yes~~, I believe this scheme may work for this grammar in particular, as using $E \rightarrow E+T$ will produce ≥3 ~~tokens in the string~~.

Ans. No, this will not work, consider the string 1+2+3

Ans. No, this scheme will NOT ~~Yes, this scheme~~ work for this grammar in particular, as using $E \rightarrow E+T$ ~~produces~~ is a left recursive grammar.

There is no way for us to check.

3. Consider the following context-free grammar:
   E → EEa | EbE | cEE | d
   S → E$

   Here "\epsilon" represents the empty string. "$" represents the end-of-input marker.

   a. What is the first-set of S, First(S)? [2]

   and E

   FIRST(E) = {c, d}

   FIRST(E) ⊆ FIRST(S) = {c, d}

   b. What is the follow-set of S, Follow(S)? [2]

   and E

   FOLLOW(E) = {a, b, c, d, $}

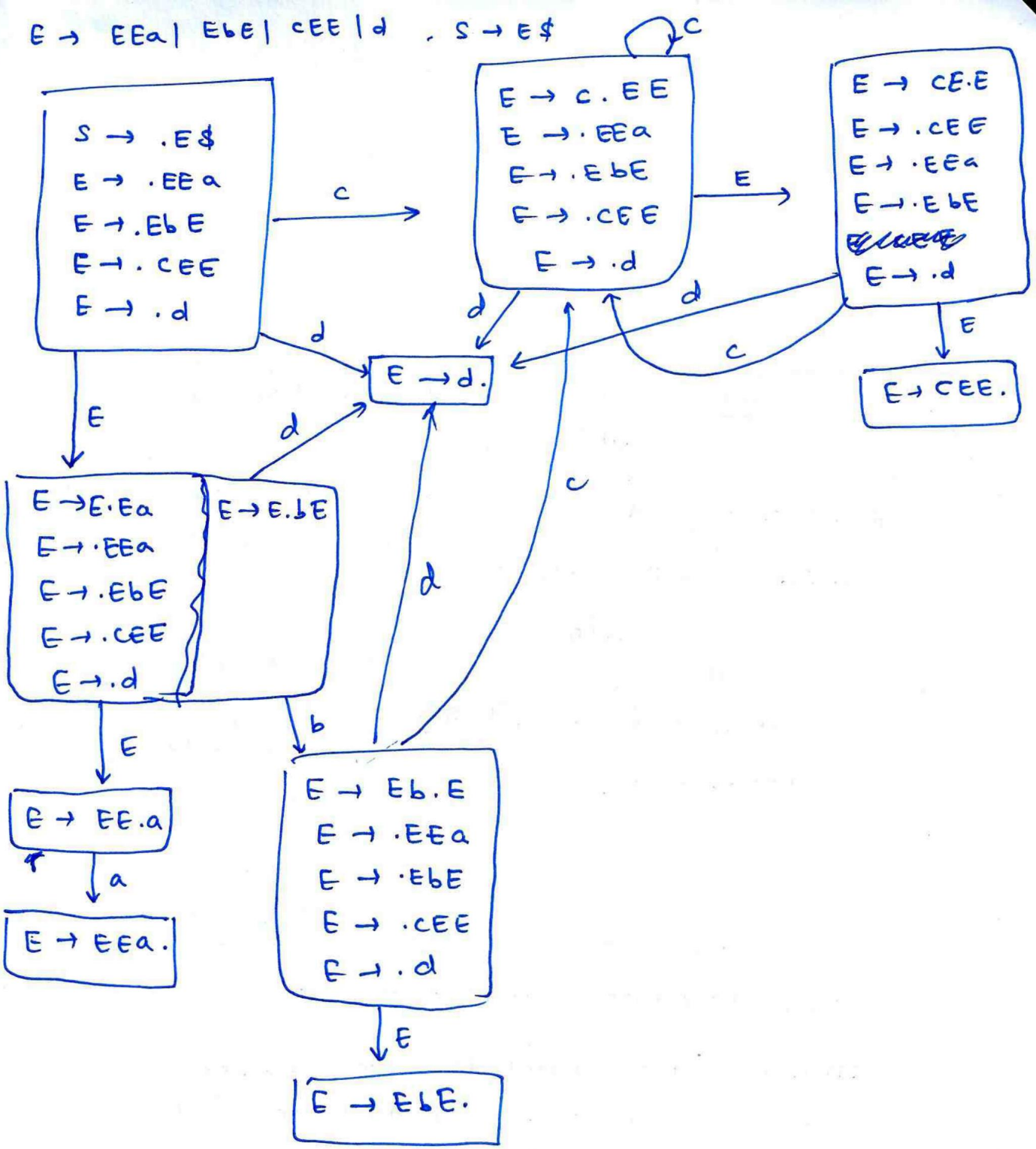
   FOLLOW(S) = { }

   c. Is this grammar LR(0)? Briefly show your working and your reasons for your answer. [6]

   LR(0) automaton on next page →

   From the LR(0) automaton, we can see that there are no shift-reduce or reduce-reduce conflicts.

   ∴ Grammar is LR(0)


E → EEa | EbE | cEE | d , S → E$
S → .E$
E → .EEa
E → .EbE
E → .cEE
E → .d
c
E → c.EE
E → .EEa
E → .EbE
E → .cEE
E → .d
E
E → cE.E
E → .cEE
E → .EEa
E → .EbE
E → .d
d
E → d.
E
E → cEE.
E → E.Ea
E → .EEa
E → .EbE
E → .cEE
E → .d
E → E.bE
E → EE.a
a
E → EEa.
E → Eb.E
E → .EEa
E → .EbE
E → .cEE
E → .d
E → EbE.

4. List one (primary) advantage and one (primary) disadvantage of LL(k) parsing over LR(k) parsing. [4]

Ans. DISADVANTAGE

LL(K) $\subsetneq$ LR(K)

i.e. the expressive power of LR(k) languages is greater than LL(k) languages (strictly).

→ So, we will be able to parse a language with more complex constructs using LR(k).

~~DISADVANTAGE~~

ADVANTAGE

The size of LL(k) parsing tables is smaller than the typical size of an LR(k) automaton, since we need to take the product of the state space with each of the lookahead tokens.

On the contrary, we have seen that in LL tables most of the entries are actually empty ~~space~~ so space-wise it is a lot more efficient.

Even with DFA compression techniques, the overhead of the cartesian product of |S| & k can be huge.

5. How are errors typically handled during the lexing phase? Answer briefly and clearly. [4]

Ans. During lexing, the identification of each token is independent, so we should NOT stop lexing & return to the programmer. instead we should continue & try to return as many (useful) errors to programmer as possible.

Therefore, during the pattern matching of a string, suppose it does not match with the "pattern" for any token, then we can take 2 approaches —→

① Either we can generate an "ERROR" token, which is then sent to the parser to deal with. This allows us to continue lexing & return as many errors to the programmer as possible (we already know program is incorrect). ~~Take a callback~~ we can implement how to deal with an error token in the parser as well.

② The other option is to see which patterns are "close" to the erroneous lexeme & try to correct the string to generate the "closest" matching token. However, this is likely to produce unexpected errors which can confuse the programmer (as the compiler is trying to second guess what the programmer meant).

6. Consider the following typing rules for expressions (e1, e2, ..) in a C-like programming language (as discussed in class).

Rule 1:

$$\frac{\vdash e_1 : int \qquad \vdash e_2 : int}{\vdash e_1 + e_2 : int}$$

Rule 2:

$$\frac{\vdash e_1 : int \qquad \vdash e_2 : int}{\vdash e_1 + e_2 : void}$$

a . Which of these two rules is sound, if any? Explain briefly [3]

Rule 1 is sound, since it agrees with the execution semantics of the language, i.e. when we compute an expression e_1+e_2, we want $V(e_1+e_2) = V(e_1) + V(e_2)$ and $V(e_1+e_2) = int$. This matches the static type rules, & so rule1 is sound.

b. Which of these two rules would accept a larger set of programs as type-correct? Explain briefly. [3]

we will be able to accept more programs using rule 1.

The reason for this is that

$$int <: void$$

So, although it will be able to ~~compute~~ typecheck e_1+e_2, if we have

$e_1 + e_2 + e_3$, e_1+e_2 is of type void

but void & int cannot be added.

So, a statement like :

$x = 1+2+3$ will not ~~get~~ pass the typechecker.

So, rule 2 will reject many valid programs.

7. To implement static scoping, we use a symbol table and the following associated helper functions and data structures:

- For implementing a symbol table, we can use a stack of scopes (assuming C89)
  - enter_scope(): start a new nested scope
  - find_symbol(x): search stack starting from top, for the first scope that contains x. Return first x found or NULL if none found
  - add_symbol(x): add a symbol to the current scope (top scope)
  - check_scope(x): true if x defined in the current scope (top scope). allows to check for double definitions.
  - exit_scope(): exit current scope

How will the compiler's implementation change if the language was instead dynamically scoped? [5]

Ans. For a dynamically scoped language, the symbol table must store the most closely binding of an identifier. This depends on the runtime execution of the program. and cannot be checked at compile time. However, the compiler can generate code to do the scoping dynamically. It can also do a "basic" check at compile time: keep an array/stack of ALL scopes. Then,

- enter_scope()
- find_symbol(x): Look through the array of scopes to find a scope which contains x.
  (It is possible that scope "may" be executed before current scope at runtime).
- add_symbol(x): Add to current scope
- check_scope(x): True if x defined in current scope.
- exit_scope: Exit current scope.

Further, for runtime codegen, compiler can instruct the machine to keep a global hashtable/table of variables & whenever the identifier (ID) is referred to, update this table so that the closest binding value is always returned on lookup(x).

8. During code generation for our toy language (as discussed in class and lecture modules), we use the following code generation implementation on a RISC machine that is emulating a one-register stack machine (with the expression value being held in register $a0). Assume that all the code generation invariants discussed in the class (and lecture modules) hold.

Code for the "caller side":	Code for the "callee side":
cgen(f(e1,e2,...,en)) = sw $fp 0($sp) addiu $sp $sp -4 cgen(en) sw $a0 0($sp) addiu $sp $sp - 4 ... cgen(e1) sw $a0 0($sp) addiu $sp $sp - 4 jal f_entry	cgen(def f(x1,x2,...,xn) = e) = f_entry: move $fp $sp sw $ra 0($sp) addiu $sp $sp -4 cgen(e) lw $ra 4($sp) addiu $sp $sp *z* lw $fp 0($sp) jr $ra

In this scheme, the responsibilities for implementing the dispatch and return of the function call

8. During code generation for our toy language (as discussed in class and lecture modules), we use the following code generation implementation on a RISC machine that is emulating a one-register stack machine (with the expression value being held in register $a0). Assume that all the code generation invariants discussed in the class (and lecture modules) hold.

Code for the "caller side":	Code for the "callee side":
cgen(f(e1,e2,...,en)) = sw $fp 0($sp) addiu $sp $sp -4 cgen(en) sw $a0 0($sp) addiu $sp $sp - 4 ... cgen(e1) sw $a0 0($sp) addiu $sp $sp - 4 jal f_entry	cgen(def f(x1,x2,...,xn) = e) = f_entry: move $fp $sp sw $ra 0($sp) addiu $sp $sp -4 cgen(e) lw $ra 4($sp) addiu $sp $sp *z* lw $fp 0($sp) jr $ra

In this scheme, the responsibilities for implementing the dispatch and return of the function call are somewhat shared between the caller and the callee. How would you change this code generation implementation such that most of the work is performed by the caller, and the minimal amount of work is performed by the callee? [5]

CALLER SIDE :

cgen(f(e1,...en)) =
sw $fp 0($sp)
addiu $sp $sp -4
cgen(en)
:
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp-4
mov $fp $sp
sw $ra 0($sp)
addiu $sp $sp-4
jal f-entry
add iu $sp $sp 2
lw $fp 0($sp)

Callee side :

f-entry :
cgen(e)
lw $ra 4($sp)
jr $ra

Invariant (additional):

→ When callee is called, return address is at the top of stack.