**CS131 Final Exam Cover Sheet (Solution)**

Instructor: Fu Song

June 5, 2017

Your name: Your email ID: @shanghaitech.edu.cn

**INSTRUCTIONS**

* You have 180 minutes (9:00-12:00) to complete the exam.
* Your exam will not be graded unless you complete the cover sheet, and turn in both this exam book and the cover sheet.
* This exam is open-book and open-notes. You may use laptops, phones and e-readers to read electronic notes, but not for computation or access to the internet for any reason.
* Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

Write on the following lines:

*“I certify that I am the person with the above name and email address. By taking this exam, I agree to all the rules set forth by the instructor.”*

SIGN your name:

|  |  |  |
| --- | --- | --- |
| Problem | Max | Points |
| 1 | 10 |  |
| 2 | 15 |  |
| 3 | 15 |  |
| 4 | 32 |  |
| 5 | 10 |  |
| 6 | 18 |  |
| 7 Extra Credits | 10 |  |
| Total | 100 |  |

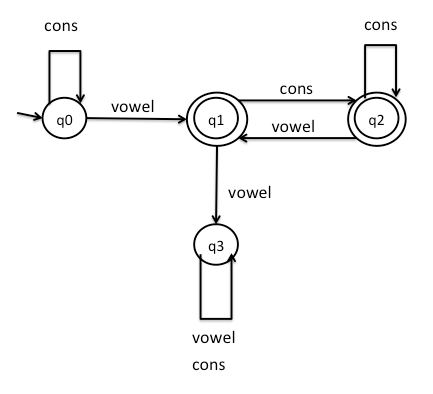
You must turn in both the cover sheet and exam book.

**Question 1 (5+5=10 points).** This question concerns the language of words. A word contains vowels and consonants and must follow these rules:

1. Every word must have at least one vowel.

2. No word can have two consecutive vowels.

**Part (a) Draw a complete deterministic finite-state machine** for the language of words as described above. Label the start state "S" and use a double circle to indicate the final state(s). You may label the edges vowel or cons (meaning consonant).



**Part (b) Write a regular expression for the language of words.** You may use vowel and cons in your regular expression (\*: zero or more, +: one or more, |: or)

There are many possible ways to do this; here are two:

* cons\* vowel (cons + vowel)\* cons\*
* cons\* vowel (cons vowel | cons+)\*

**Question 2 (7+8=15 points).** Below is a context-free grammar for a language of assignments that includes arrays:

1. stmtList 🡪 stmt stmtList

2. stmtList 🡪ε

3. stmt 🡪 ID = exp ;

4. array 🡪[ rowList ]

5. rowList 🡪nonEmpty

6. rowList 🡪 ε

7. nonEmpty 🡪 row moreRows

8. moreRows 🡪; nonEmpty

9. moreRows 🡪ε

10. row 🡪exp more

11. more 🡪, row

12. more 🡪 ε

13. exp 🡪term tail

14. tail 🡪+ term tail

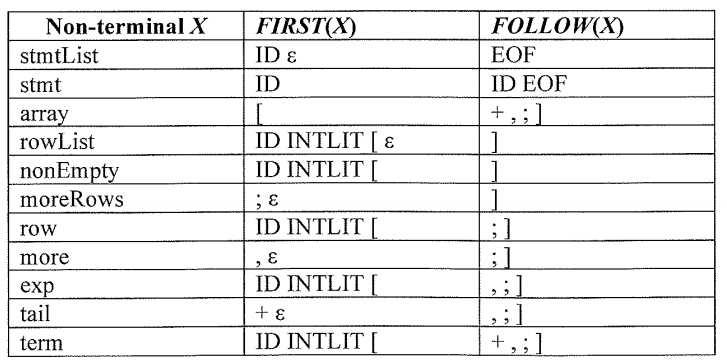
15. tail 🡪ε

16. term 🡪ID

17. term 🡪INTLIT

18. term 🡪 array

**Part (a).** Fill the FIRST and FOLLOW sets for all of the non-terminals:



**Part(b).** Fill in the parse table below using the numbers of the grammar rules rather than the rules themselves. Is the grammar LL(1)?

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ID | INTLIT | = | + | ; | . | [ | ] | EOF |
| stmtList | 1 |  |  |  |  |  |  |  | 2 |
| stmt | 3 |  |  |  |  |  |  |  |  |
| array |  |  |  |  |  |  | 4 |  |  |
| rowList | 5 | 5 |  |  |  |  | 5 | 6 |  |
| nonEmpty | 7 | 7 |  |  |  |  | 7 |  |  |
| moreRows |  |  |  |  | 8 |  |  | 9 |  |
| row | 10 | 10 |  |  |  |  | 10 |  |  |
| more |  |  |  |  | 12 | 11 |  | 12 |  |
| exp | 13 | 13 |  |  |  |  | 13 |  |  |
| tail |  |  |  | 14 | 15 | 15 |  | 15 |  |
| term | 16 | 17 |  |  |  |  | 18 |  |  |

**Question 3 (7+8=15 Points).** Consider the following grammar:

1. S🡪 B
2. B🡪 BE+
3. B🡪 -E
4. E🡪 e
5. E🡪 Ee
6. E🡪ε

**Part(a**) Write the LR(1) automaton.

**S🡪.B, $**

**B🡪. BE+, $/e/+**

**B🡪. –E, $/e/+**

0 1

**E🡪 e., e/+**

B

e

-

**B🡪 –.E, $/e/+**

**E🡪. e, $/e/+ E🡪. Ee, $/e/+**

**E🡪. , $/e/+**

**S🡪 B., $**

**B🡪B.E+, $/e/+**

**E🡪. e, e/+ E🡪. Ee, e/+**

**E🡪., e/+**

2 3 $ accept

E E

4 5

**B🡪 –E., $/e/+**

**E🡪 E.e, $/e/+**

**B🡪 BE.+, $/e/+**

**E🡪 E.e , e/+**

e

e e

**E🡪 Ee., $/e/+**

6 7

**E🡪 Ee., e/+**

+

8 9

**B🡪 BE+., $/e/+**

**E🡪 e., $/e/+**

**Part(b).** Write the action table and goto table for LR(1) parsing

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| State | Action | | | | Goto | | |
| + | - | e | $ | S | B | E |
| 0 |  | S2 |  |  |  | 3 |  |
| 1 | R4 |  | R4 |  |  |  |  |
| 2 | R6 |  | S9,R6 | R6 |  |  | 4 |
| 3 | R6 |  | S1R6 | accept |  |  | 5 |
| 4 | R3 |  | S6R3 | R3 |  |  |  |
| 5 | S8 |  | S7 |  |  |  |  |
| 6 | R5 |  | R5 | R5 |  |  |  |
| 7 | R5 |  | R5 |  |  |  |  |
| 8 | R2 |  | R2 | R2 |  |  |  |
| 9 | R4 |  | R4 | R4 |  |  |  |

**Question 4 (12+20=32 Points).** Consider an extension of Cool to support arrays of objects. We introduce an Array class that inherits from Object. Other classes cannot inherit from the Array class. We introduce four new expressions for manipulating Cool arrays:

e ::= new Array[e]

| e1[e2]

| e1[e2] ← e3

| foreach vi , ve in e1 do e2

Subexpressions are evaluated left-to-right (e.g., e1 before e2). The first expression form creates a new array of size e. The array initially holds e separate copies of new Object. The size must be non-negative at runtime to avoid an exception. The second expression form reads from array e1 at index e2, returning the object stored there. The third writes to array e1 at index e2 the value e3 (and returns e3). For reads and writes, the index must be between 1 and the size of the array at runtime to avoid an exception. The final expression executes e2 for every element in array e1 with variable name vi bound to the that element’s index and variable name ve bound to that element’s value. Each element is considered in ascending order starting from 1. For example, this code:

let arr : Array <- new Array[3] in

arr[3] <- 5309;

arr[1] <- 867;

arr[2] <- "unicorn";

foreach i, elt in arr do {

out\_string("element ") ; out\_int(i) ;

out\_string(" is ");

case elt of

n : Int => out\_int(n) ;

s : String => out\_string(s) ;

esac;

out\_string("\n");

} ;

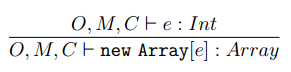
Produces:

element 1 is 867

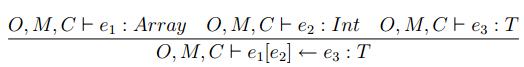
element 2 is unicorn

element 3 is 5309

**Part(a).** Give typing rules for the three new array expressions. Be as permissive as possible without permitting any unsafe programs. The type rule of new Array[e] is given as an example, you need to write the other three typing rules.



C:\Users\FUSONG~1\AppData\Local\Temp\1495035109(1).png

C:\Users\FUSONG~1\AppData\Local\Temp\1495035005(1).png

**Part(b).** We extend our notion of Cool values v to include array objects and an “ArrayException” keyword report any array problem at run-time. If there is any error to access array, return the value “ArrayException”

v ::= void

| X(a1 = l1, … , an = ln)

| Array(l1,…, ln)

| ArrayException

Give the new operational semantics rules. Write only those rules where all subexpressions e, e1 and e2 return. Do denotes the default value of the instance of the class Object, newloc(S) creates a new location which has not yet been used in S. AE is the shortcut of ArrayException. Two operational semantics rules are given as examples. You have to write the other **five** operational semantics rules.

so, E, S1 |- e1 : a, S2

a = Array(l1,…,ln)

so, E, S2 |- e2 : i, S3

1<= i<=n

v = S3[li]

so, E, S1 |- e1[e2]: v, S3

so, E, S1 |- e : n, S2

n ≤ 0

so, E, S1 |- new Array[e]: AE, S2

so, E, S1 |- e : n, S2

n > 0

l1,…,ln = newloc(S1),…,newloc(S1)

S3=S2[l1🡪Do,…,ln🡪Do]

v = Array(l1,…,ln)

so, E, S1 |- new Array[e]: v, S3

so, E, S1 |- e1 : a, S2

a = Array(l1,…,ln)

so, E, S2 |- e2 : i, S3

i<1 ori>n

so, E, S1 |- e1[e2]: AE, S3

so, E, S1 |- e1 : a, S2

a = Array(l1,…,ln)

so, E, S2 |- e2 : i, S3

so, E, S3 |- e3 : v, S4

1<= i<=n

S5 = S4[li🡪v]

so, E, S1 |- e1[e2]🡨e3: v, S5

so, E, S1 |- e1 : a, S2

a = Array(l1,…,ln)

so, E, S2 |- e2 : i, S3

so, E, S3 |- e3 : v, S4

i <1 ori>n

so, E, S1 |- e1[e2]🡨e3:AE, S4

or

so, E, S1 |- e1 : a, S2

a = Array(l1,…,ln)

lvi=newloc(S2)

so, E[vi🡪lvi, ve🡪l1],S2[lvi🡪1]|-e2:c1,S3

so, E[vi🡪lvi, ve🡪l2],S3[lvi🡪2]|-e2:c2,S4

…

so, E[vi🡪lvi, ve🡪ln],Sn+1[lvi🡪n]|-e2:cn,Sn+2

so, E, S1 |- foreach vi,ve in e1 do e2:void, Sn+2

so, E, S1 |- e1 : a, S2

a = Array(l1,…,ln)

so, E, S2 |- e2 : i, S3

i <1 ori>n

so, E, S1 |- e1[e2]🡨e3:AE, S3

**Question 5 (10 Points).** Consider the following grammar:

S🡪 if C then Stmts Rest

Rest🡪elseif C then Stmts Rest

| else Stmts endif

C🡪 id relop id

Example:

if x<y then

Stmts

elseif a>b then

Stmts

else

Stmts

endif

Provide the semantic actions that generate code for the above control flow construct. The generated code should be available in the code attribute of S (i.e., S.code). Assume that semantic actions specified by other productions not shown here will place the code generated for Stmts in Stmt.code, id.place is the name that holds the value of id, and newlabel() for creating a new label.

C🡪 id relop id { turelabel = newlabel();

C.falselabel = newlabel();

C.code = gen(“if” id1.place relop.code id2.place “jmp”

truelabel) || gen(“jmp” C.falselabel) || gen(truelbael“:”)

}

S🡪 if C then Stmts {

Rest.iflaselabel = C.falselabel;

Rest.iexit = newlabel();

Rest.icode = C.code || Stmts.code || gen(“jmp” Rest.iexit)

}

Rest { S.code = Rest.scode }

Rest1🡪elseif C then Stmts {

Rest2.icode =Rest1.icode||gen(Rest1.ifalselabel”:”)

|| C.code || Stmts.code || gen(“jmp” Rest1.iexit);

Rest2.ifalselabel = C.falselabel;

Rest2.iexit = Rest1.iexit

}

Rest2 {Rest1.scode = Rest2.scode }

Rest🡪 else Stmts endif {

Rest.scode = Rest.icode || gen(rest.ifalselabel”:”)

|| Stmts.code || gen(Rest.iexit”:”)

}

**Question 6 (8+10 =18 Points).** Consider the following code sequence, where a and b are some integers:

X = a

Y = b

If X<Y Goto L1

Z = X + Y

X = Y

Goto L2

L1:

Z = X-Y

X =Y

L2:

Print(X)

Print(Z)

**Part(a).** Construct the control flow graph of the above program.

X = a

Y = b

if X<Y Goto L1

X

X, Y

X, Y

X, Y

Z = X+Y

X = Y

L1: Z = X-Y

X =Y

Y, Z

Y, Z

X, Z

X, Z

L2: Print(X)

Print(Z)

Z

**Part(b).** Perform liveness analysis for all variables (directly label to the control flow graph). Suppose X and Z dead after Print(Y).

**Question 7 (10 Points, extra credits).** Consider the extension of Cool to support arrays of objects in Question 4. Following the approach described in class, we decide to layout array objects as follows:

|  |  |
| --- | --- |
| 0 | Type tag |
| 4 | Object size |
| 8 | Array element 1 pointer |
| 12 | Array element 2 pointer |
| … | … |
| 4 + 4n | Array element n pointer |

Give the stack machine code generation rule for cgen(e1[e2] ← e3). You have to return ArrayException when a runtime error occurs. You can use registers: $a0, $t0,…,$t2. The result of each expression is stored in $a0. The following instructions you may need: push r/imm, pop r, ld r1 offset[r2] (load value from memory r2+offset into r1), st r1 offset[r2] (store value of r1 into memory r2+offset), add r1 r2/imm(r1=r1+r2/imm), sub r1 r2/imm (r1=r1-r2/imm), mul r1 r2/imm (r1=r1 \*r2/imm), r1:=r2/imm (assignment r1=r2/imm), jmp label (jump), if r1 rel r2/imm (rel can be >,<,>=,<=,=,!=)

cgen(e1[e2] <- e3) =

cgen(e1)

push $a0 // save array into the stack

cgen(e2)

$t0 := $a0 # t0 holds index

pop $t1 # t1 holds array

ld $t2 4[$t1] # load array object size into $t2

sub $t2 2 # number of elements

if $t0 < 1 goto err # index too low

if $t0 > $t2 goto err # index too high

add $t0 1

mul $t0 4

push $t0

push $t1

cgen(e­3)

pop $t1

pop $t0

st $a0 $t0[$t1] # t0+t1 points to e1[e2]

jmp end

err:

$a0:= ArrayException

end: