COL728 Major Test

Viraj Agashe

TOTAL POINTS

54 / 60

QUESTION 1

1 1 Code Generation 8.5 / 10

Correct

✓ **+ 3.5 pts** **assign -- compute+store**

✓ **+ 3 pts** **return -- compute+return to caller**

+ 1.5 pts def -- changes for S (and return)

✓ **+ 2 pts** **seq**

+ 0 pts Incorrect/not attempted

`cgen` for `def id(ARGS) = S` is missing -- the original language had `def id(ARGS) = E` (note: E -> S).

QUESTION 2

2 2a Semantic rule what is lxi 2 / 2

✓ **+ 2 pts** **Correct**

+ 0 pts Incorrect / Unattempted

QUESTION 3

3 2b Significance of new locations for lxi 2 / 2

✓ **+ 2 pts** **Correct**

+ 1.5 pts Partially correct

+ 0 pts Incorrect / Unattempted

QUESTION 4

4 2ci Correctness of semantics 3 / 3

✓ **+ 3 pts** **Correct**

+ 2 pts Partially correct

+ 0 pts Incorrect / Unattempted

QUESTION 5

5 2cii Deletion of new locations 1.5 / 4

+ 4 pts Claims harder with solid arguments

+ 2.5 pts Claims easier with solid arguments

✓ **+ 1.5 pts** **Partially valid arguments for either**

+ 0 pts Incorrect arguments / Unattempted

Indeed, otherwise the compiler may choose to heap allocate everything (which makes it very inefficient) or heap allocate stuff whose locations are leaky (through a complex memory-leak analysis). In any way, it makes the job of the compiler harder, do you agree?

QUESTION 6

6 3 Example of Phase Ordering Problem 5 / 5

+ 0 pts Not attempted

Correct

✓ **+ 2 pts** **Valid passes**

✓ **+ 1 pts** **Explanation of passes.**

✓ **+ 2 pts** **Valid example program**

QUESTION 7

7 4a Pointer adjustment 3 / 3

✓ **+ 3 pts** **Correct**

+ 0 pts Incorrect

QUESTION 8

8 4b Pointer readjustment 4 / 4

✓ **+ 4 pts** **Correct**

+ 0 pts Incorrect/not attempted

QUESTION 9

9 5 scope, lifetime, activation tree 3 / 5

✓ **+ 3 pts** **Correct example program**

+ 1.5 pts Partially correct example program

+ 2 pts Correct activation tree for the example program

+ 1 pts Partially correct activation tree for the example program

+ **0 pts** Incorect/Unattempted

+ **0 pts** Incorrect/Unattempted activation tree

QUESTION 10

10 6a When is alignment necessary 3 / 3

✓ **+ 3 pts Correct**

+ **0 pts** Unattempted/Incorrect

+ **1.5 pts** Partially Correct

QUESTION 11

11 6b When is alignment good for performance 3 / 3

✓ **+ 3 pts Correct**

+ **0 pts** Incorrect/Unattempted

+ **1.5 pts** Partially Correct

QUESTION 12

12 7 DFA for null-pointer check 16 / 16

✓ **+ 4 pts Correct Domain of DFA Values**

✓ **+ 4 pts Correct Meet operator**

✓ **+ 4 pts Correct Transfer Function Description**

✓ **+ 4 pts Correct Initialization**

+ **0 pts** Incorrect / Not Attempted

VIRAJ AGASHE
2020CS10567

COL728 Major Exam
Compiler Design
Sem I, 2022-23

Answer all seven questions Max. Marks: 60

1. To understand code generation, the language that was considered in class supported only immutable variables, i.e., variables that can only be assigned (mutated) once throughout the program execution. The following grammar describes the programming language (with immutable variables) that was discussed in class:

P → D; P | D

D → def id(ARGS) = E
ARGS → id, ARGS | id
E → int | id | if E1 = E2 then E3 else E4 | E1 + E2 | E1 - E2 | id(E1,...,En)

Recall that we also discussed a recursive program written in this language that computes the n-th fibonacci number. We also demonstrated an implementation for the "cgen" function for this language in class.

Consider the following grammar that is slightly different from the grammar above:

P → D; P | D
D → def id(ARGS) = S
ARGS → id, ARGS | id
E → int | id | if E1 = E2 then E3 else E4 | E1 + E2 | E1 - E2 | id(E1,...,En)
S → id := E | return E | S; S

Thus, the primary difference between this new grammar and the original grammar is that: the new grammar supports "statements" (represented by the non-terminal "S"). A statement is formed through the assignment operator ("id := E") or a return keyword (to return a value at the end of the body of the function which is now a statement). Multiple statements can also be sequenced (using the semi-colon operator) to form a single compound statement. The semantics of statements in this language are similar to the semantics of a typical imperative language like C/Java/etc.

How will you change the "cgen" function to implement the semantics of the new language. In particular, demonstrate the "cgen" implementation for the assignment operator ("id := E"), the "return E" statement and the sequence operator ("S;S"). [10]

Ans. Code generation

(I) cgen (id := e)

- Since we assume the only ids are the function parameters/ arguments, we can access them using the AR and the $fp.

For convenience let us call the variable id as x_i (i.e. the i^{th} argument of the function body we are in).

So, the codegen function.

~~cgen (id := e) →~~ cgen(e) # val of e now in $a0
sw $a0, 0($sp)
addiu $sp, $sp, -4
sw

$$cgen(x_i := e) \rightarrow cgen(e) \quad \# \text{ val now in } \$a0$$
sw $a0, 4i($fp) # Store value of e in fp + 4i

(II) cgen (return E)

To support return statements, we do not need to do much. We already have by our invariants that the result of a expression after cgen is stored in $a0. So we just need to ensure that we store the value of E in the register $a0.

[Since return is guaranteed to be at the end, we will always get $a0 → result of function return].

cgen(return e) → cgen(e)

(III) cgen (S; S)

cgen(S_1; S_2) → cgen(S_1)
cgen(S_2)

2. Consider the formal operational semantics for the function call dispatch in an imperative programming language, as discussed in class, and as shown below:

```
s0,E,S |- e1 : v1,S1
s0,E,S1 |- e2 : v2,S2
...
s0,E,S(n-1) |- en : vn,Sn
s0,E,Sn |- e0 : v0,S(n+1)
v0 = X(a1=l1, ..., am=lm)
impl(X,f) = (x1,..,xn, ebody)
lxi = newloc(S(n+1)) for i=1..n
E' = [a1:l1, ..., am:lm][x1/lx1,...,xn/lxn]
S(n+2) = S(n+1)[v1/lx1,...,vn/lxn]
v0,E',S(n+2) |- ebody:v,S(n+3)
------------------------------------------- [dispatch]
s0,E,S |- e0.f(e1,...,en) : v, S(n+3)
```

The "newloc" operator returns new locations that are not already mapped-to in the store (that is passed as an argument). In this semantic rule, the following pre-condition returns "n" different new locations that are all mutually distinct, and also distinct from any existing location in S(n+1).

```
lxi = newloc(S(n+1)) for i=1..n
```

a. What does "lxi" represent here? [2]

Ans. Here, lxi represents the address of the i^{th} argument x_i of the function f implemented for the class (type) X. It is mapped to x_i in the environment E', and its value maps to the valuation of the ith argument V_i passed to the function, i.e. $E'[x_i] : lx_i$, $S(n+2)[lx_i] : V_i$.

b. What is the significance of creating new locations for "lxi"? [2]

Ans. The creation of new locations for lxi signifies the "~~pass~~ by value" semantics of the language — this means that the formal arguments for the function will actually be stored in the new memory locations pointed to by lxi's. It is also important as the arguments are expressions themselves and therefore the function body ebody will need access to their evaluated values. Therefore, we need to store these values in new memory locations available only in the scope of ebody.

c. The semantics do not delete the locations assigned to "lxi" at the completion of the function call dispatch. Let's try and understand the implications of this choice of semantics. Consider what happens if the programming language also supports the "&" (address-of) operator, just like it is supported in C/C++ languages. In this case:

i. Would this choice of semantics produce the expected results on the execution of a function dispatch (as we expect its behaviour in languages like C/C++)? [3]

Ans. Yes, this choice of semantics would produce the expected results, since the memory locations are not being deleted, so formal arguments. to the function can be addressed using the & operator.

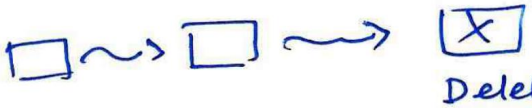
---

ii. Does this choice of semantics make the job of the compiler easier or harder (compared to another choice of semantics where these locations are deleted at the completion of the function call dispatch). Briefly explain why or why not. [4]

Ans. I believe this choice of not deleting the locations makes the job of the compiler easier. ~~The compiler has to check whether~~

If the compiler deleted the locations assigned to lxi, then we would need to have separate code generated whenever we return an address to one of the newly allocated locations from within the function body.

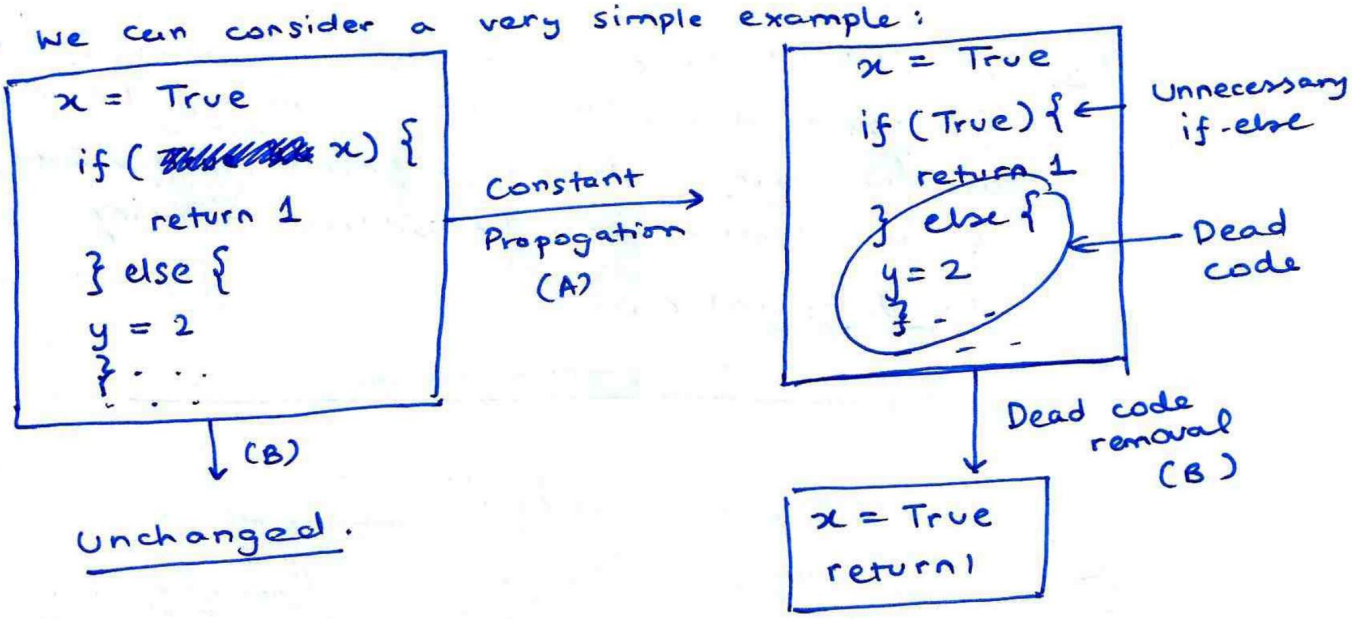
If we did not have this separate code, then pointer chasing would become impossible / would give errors



On the other hand, the existing semantics are able to handle this.

3. What is an example of a sequence of program transformation passes A and B where the second transformation (B) improves the program only after the first transformation (A) has already been performed. In other words, the transformation B would leave the program unchanged if it was applied first. Briefly explain the nature of the transformation passes "A" and "B" and also show the original program and its transformations (in each case). [5]

Ans. We can consider a very simple example:



From the above example, we can see that only after constant propogation has been applied, we see that the 'if' condition is always true and the else code is unreachable.

∴ We can now apply dead code removal to remove it (and unnecessary if-else)

If we directly applied dead code elimination first, it would not have known the 'else' code block is unreachable and would have left it unchanged.

Nature of passes:

1. Constant Propogation: Whenever we know that the variable is a constant, we replace a reference to it by the constant itself. This is done by DF.A techniques.

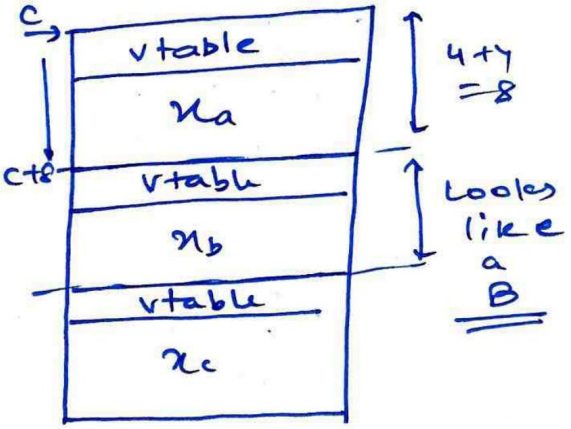
eg. $x = 3$ ⟶ $x = 3$
$y = x + 2$ $y = 5$

2. Dead code removal: Code whose unreachability is known at compile time can be removed to save on the no. of instructions.

4. In the context of "code generation for object oriented languages", briefly explain the following terms:

a. Pointer adjustment [3]

Ans. In the case of multiple inheritance say A → C ← B, then suppose the object is laid out as in the figure. Now when C is called in the context of B (which it can be due to inheritance, C <: B)

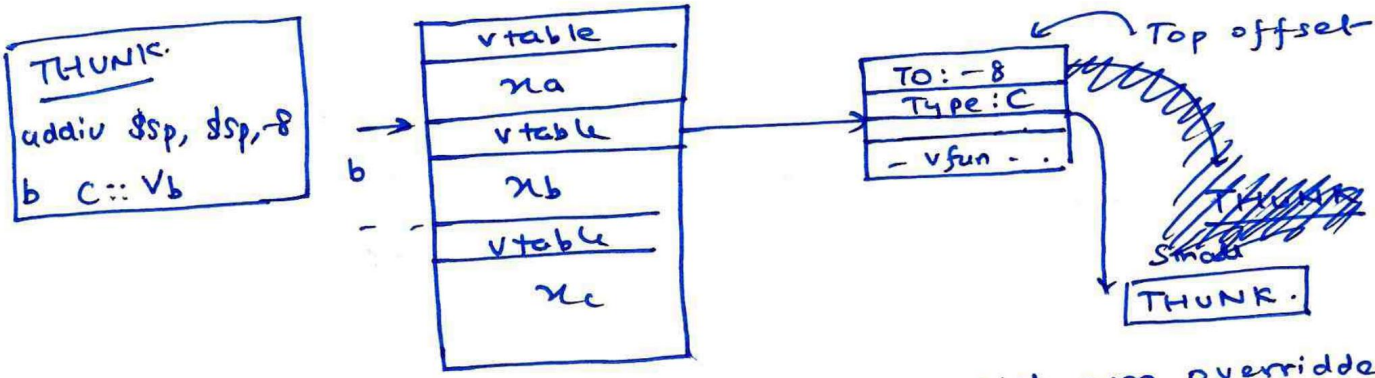


The object does NOT look like a B object. So, in this case the pointer to the object c is adjusted s.t. we add the offset of the size of vtable of a & x_a (in this case 8). So, by changing the pointer to C+8 (or C−8 depending on direction of stack growth), we get a pointer to an object which looks like and can be used as a B by the compiler. This is pointer adjustment.

b. Pointer readjustment [4]

Ans. Now, suppose that we want to get back the object of type C from B (eg. we call a virtual function overridden by C)

In such a scenario,



Suppose we have called b.Vb() which was overridden by C. Then, we want to call C's vb(). However, for the call to work the object must be laid out as a 'C' (appear to the function). For this purpose we keep a small code called "thunk" which first uses the value of the top offset T.O. stored in the vtable to perform pointer readjustment

C = b − 8

To get back an object of type C from b. so that the call to C::Vb() works.

5. Show an example C program where the scope and the lifetime of a variable are different. Draw the activation tree for that program. [5]

```
A* myfun() {
    A* a = new A();
    return a;
}
int main() {
    A* val = myfun();
    return val->label;
}
```

Consider the ~~C++~~ C++ program here.

The variable a of type A* is created within the scope of the function body of myfun().

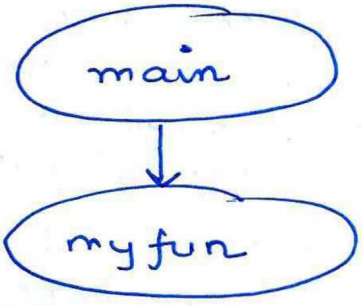
However, since we use the new operator, the object A is allocated on the heap & not the stack.

Therefore, the lifetime of the variable a is the point till which it is deleted. However, since we never call the delete keyword explicitly, the lifetime of a is the lifetime of the entire program (after initialization of a).

→ Therefore, even though the scope of the ~~program~~ variable a is only its scope within the function body, the actual lifetime of the variable at runtime is different.

→ The variable val created in main is nothing but a pointer to the original object 'a' created on the heap. So we can clearly see that a outlives its scope.

Activation tree



6. The programming languages and compilers often ensure that addresses of multi-byte data-types (e.g., integers or floats or doubles in C) are aligned to some power of two (e.g., four-byte or eight-byte aligned).

a. What is an example of a situation where such alignment is necessary? [3]

Ans. In many MIPS (RISC) architectures, it is compulsory for multi-byte memory accesses (eg. word addressing) ~~is strictly allowed~~ to be 4/8/.. byte aligned, i.e. there is a hardware restriction which disallows non-aligned memory accesses of multiple-bytes. In such a case, it is important for datatypes like ints / floats / etc. to have 4/8 byte aligned addresses.

eg. class foo {
int a;
char b;
int c; }

← Since HW only allows aligned accesses, must give padding & lay out in aligned manner.

b. What is an example of a situation where such alignment is not necessary but preferable because it provides better performance? [3]

Ans. For x86 architectures, non-aligned addresses are allowed. However, hardware is still specialized such that aligned memory accesses are faster than non-aligned accesses — this is a hardware quirk which makes the hardware logic faster, & easier to implement. Therefore it is possible to read all 4/8 bytes of a multibyte access at once (due to bandwidth of the Buses etc), in a case with alligned accesses.

Even though not necessary, alignment is FASTER. due to above reasons and desirable. ~~FASTER~~

eg. class foo {
int a;
char b;
int c;
}

While laying out object we could have kept c laid out contiguously with b, however that would slow down access of b as it would not be aligned.

So we give padding of 3 bytes.
to improve efficiency at cost of space.

7. I am interested in checking the following program property: "the return value of a call to the malloc() function must always be checked against null before it is de-referenced".

The following program does not satisfy this property:

```
%r1 = malloc(100);
%r2 = load %r1
```

On the other hand, the following program satisfies this property:

```
%r1 = malloc(100)
beq %r1, 0, .Lout_of_memory
%r2 = load %r1
branch .exit
.Lout_of_memory:
print("out of memory")
.exit:
Return
```

Here, "beq" is a conditional branch that performs the branch only if the first two operands are equal. Assume that the "beq" opcode is the only way to check if a value is null or not (as it is used in the example above.

Design a DFA (data flow analysis) to check this property in any arbitrary program. You will need to specify:

1. The values of your DFA [4]
2. The meet operator [4]
3. The transfer functions of the different IR instructions [4]
4. The initial values (to initialize the fixed-point algorithm) [4]

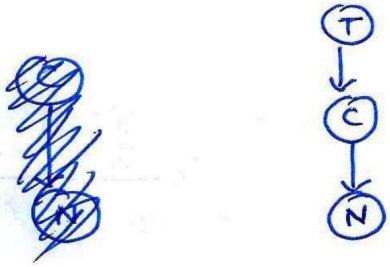
Sol. We will perform a forward DFA to check this property. We define our DFA as follows:

1. Values of DFA

We will keep only 3 values in our DFA :

Val	Interpretation
C	Checked for NULL
N	Not checked for NULL
T	Not reached so far

Partial order



2. Meet : The meet operator for my DFA is defined as :

$$C(s, r, in) = \underset{\forall p}{glb}\left(C(P_i, r, out)\right)$$

Greatest lower bound in the DFA partial order

Here, C is the function which returns if the

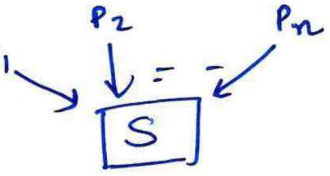
C(S, r, ~~out~~)

~~reference~~ malloced variable r has been checked for NULL at a program point S (y = in/out)

↑ Statement .

3. Transfer function. / Rules

1. $C(s, r, in) = \underset{\forall p}{glb}\left(C(P_i, r, out)\right)$



where P_is are in-neighbours of S

(This makes sense because if any path, %r has not been checked ⇒ not checked in general!)

~~C(s, r, out)~~

~~3. C(%r = ..., r, out) = C~~

2. $C(\%r = malloc(), r, out) = N$

3. $C(\%r = \ldots, r, out) = C$ — Assignment other than malloc

4. $C(\%r = f(\ldots), r, out) = N$ (May be function returns malloced NULL)

~~5. C(~~

5. $C(beq\ \%r, 1, .Lout\ldots, r, out) = C$

6. $C(S, r, out) = C(S, r, in)$

7. $C(start, r, out) = N$ (Boundary condition)

4. To initialize the fixed point algorithm,
we initialize all values to T (not reached so far)
(we choose the topmost value of the P.O. since in the DFA values can only move down the partial order).

→ By the boundary condition,
((START, r, OUT) will be N and then
we can proceed with the fixed point algo by applying rules in order.
