COL728 Minor2 Exam
Compiler Design
Sem II, 2017-18

Answer all 5 questions Max. Marks: 20

1. Short questions

a. Give an example of a program that is not a legal program if we assume static scoping, but is a legal program if we assume dynamic scoping. i.e., the same program should be well-scoped if we assume dynamic scoping but not well-scoped if we assume static scoping. Use a C-like language syntax [2.5].

E.g

```
Void foo()
{
  Int a = 5;
  bar();
}
Void bar()
{
  printf("%d", a);
}
Int main()
{
  foo();
}
```

Any correct example depicting binding of a variable to most recent declaration and no declaration in function or class scope gets full marks.

1b. Give an example to show that "~~stronger~~ static type system may force the programmer to write a slower program implementation for the same program logic". In your example, you need to show two languages, one with a weaker type system than the other. Show that the

same program logic may be slower to implement in the second language than in the first language. By “slower to implement”, we mean that the runtime of the implemented program is slower. The relative slowness of the implementation should be directly related to the stronger type system. In other words, the stronger type system should force the programmer to write a slower implementation.

Ideally, you should select from well-known languages (to make the comparison). If you decide to describe your own language (with its type system), the description needs to be very clear. [2]

One such case is:

```
Class A {
 …
 A getMySelf() { return *this; }
};
Class B : public A {
 …
};

Int main()
{
  B *ob = new B();
  B *ob2 = ob.getMySelf();
}
```

This is a C++ program which on compiling fails because of static type checking (returned object is assumed to be of type A whereas it is of type B). To implement this logic either returned object is to be dynamically casted or templates needs to be used which both slows code down by including more instructions for type checking and type casting.

Whereas if the language involved dynamic type checking (or no type checking) then dynamic type of ob->getMySelf() is “B” and that of *ob2 is also “B”, so a language with dynamic type checking will allow this kind of program logic.

Another example: mutable vs. immutable types. Some programming languages support only immutable types. In such programming languages, any modification to an existing object needs to be represented by a new object (potentially involving copying the original object). E.g., OCaml vs. C. OCaml largely has immutable types and so lists in OCaml are immutable. If I want to append an element to a list, I will need to create a new list (by potentially copying elements in the old list). On the other hand, in C, we can simply achieve this by pointer manipulation at the list head.

Any example showing program logic changed to slowed down version to pass static type checking will yield full marks.

1c. Consider the following two versions of the typing rule for the assignment statement in a C-like imperative language:

O |- x:T1
O |- e1:T1
---------- [Assignment-rule-1]
O |- x=e1 : T1

O|-x:T0
O|-e1:T1
T1 <=T0
---------- [Assignment-rule-2]
|- x=e1 : T1

Give an example of a program that would be type-incorrect if one rule is used, but would be type-correct if the other rule is used. Mention which of the rules would cause that example program to be type-incorrect, and which of the rules would cause the same example program to be type-correct. [2.5]

```
Class A {
  ….
};
Class B : public A {
 ….
};
Int main()
{
  A *ob = new B();
}
```

Any such example involving subtyping to give type mismatch error in first rule and correct by second rule with proper explanation gets full marks. You can't cause rule one to be correct and rule two to fail as every type is a subtype of itself. Partial marks will be given in case of partially true explainations.

1d. Explain how cascading errors can be avoided during type checking by defining a dummy type called "No_type", such that No_type <= T for all types T. Use an example to demonstrate this. [3]

If there is an type-mismatch (or undeclared identifier) used in a mathematical calculation then an error will be thrown for every operator in that operation sequence, this is referred as “cascading errors”. This can be solved by declaring the undeclared identifier as of type No_type just before its first use and throwing error once. This No_type must be made subtype of all other types for stopping cascading errors because only then type checking rule (2nd rule shown in previous question) will match and no error will happen.

e.g.

```
Int foo()
{
void (*foo)(int a);  //function pointer
 int x, y, z;
...
 x = x + y + (z + foo);
 return x;
}
```

In this program, without No_type, error will occur thrice in line x = x + y + (z + foo).
If we instead declare the result of (z + foo) as No_type then this statement will not have any error as well as return will also not have error as No_type is a subtype of int. The error will be raised only once when we try and type-check the expression (z + foo).

2 marks for correct explanation
1 mark for correct example

2. Consider the following programming language grammar (same as that 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)

Let NT(e) be the number of temporaries needed to evaluate e (as discussed in class).

a. How does performing this computation of NT(e) allows faster code generation on a stack machine? [1]

If we know NT(e) then we can pre-allocate space on stack at function entry, accessing it using its offset from stack/frame pointer, thereby avoiding use of subtract and move instruction on every variable declaration. This reduces code length as well as better runtime.

1 mark for correct explaination

2b. Write the code-generation procedure for function dispatch (caller and callee).

cgen(def f(x1,x2,...,xn) = e) = ?

cgen(f(e1,e2,...,en)) = ?

You can use the same (or different) code generation strategy as discussed in class, but something that takes into account the calculation of NT(e). In either case, clearly specify the code-generation strategy: e.g., what are the invariants, how you implement them, etc. [4]

Invariant maintained by the strategy: “contents of the stack are preserved after a function call i.e. value of stack before and after a function call occurs, remain same”
Another thing to note is that this strategy is generating code for a machine which has a stack and an accumulator register. Every expression’s value is stored in this accumulator register after calculation.

NT(e) refers to number of local variables or temporaries whose value needs to be stored in stack while calculating value of expression, e. If we know this value beforehand then this value can be used to preallocate space on stack and thereby removing overhead of pushing and popping the value on stack each time a temporary space is needed. NT(e) can be calculated using recursive rules like:
NT(int) = NT(id) = 0; NT(e1+e2) = max(NT(e1), 1+NT(e2));
NT(if e1==e2 then e3 else e4) = max(NT(e1), 1+NT(e2), NT(e3), 1+NT(e4));
NT(id(e1, e2, ... en) = max(NT(e1), NT(e2), … NT(en));
And so on…

To generate code by using NT values we can pass nt as an argument to cgen function which refers to location of next available termorary on the stack.
Our cgen function will not use stack pointer as we can access temporaries using fixed distance from frame pointer.
Activation record for a function will now contain all arguments and return address AND the space for all temporaries required to compute the function body.
So our cgen function for callee or function definition is:

cgen for caller remains identical as without using NT (no change except pass around nt)

```
 cgen(f(e1,e2,...,en), nt) =
sw $fp 0($sp)
addiu $sp $sp -4
cgen(en, nt)
sw $a0 0($sp)
addiu $sp $sp - 4
...
cgen(e1, nt)
sw $a0 0($sp)
addiu $sp $sp - 4
jal f_entry
 cgen(def f(x1,x2,...,xn) = e, nt) =  //this nt is not used!
f_entry:
move $fp $sp
```

```
sw $ra 0($sp)
addiu $sp $sp -4
addiu $sp $sp -NT(e)*4  //allocate additional space on the stack for the temporaries
cgen(e)
lw $ra 4($sp)
addiu $sp $sp z  //z = 4*n+8+NT(e)*4
lw $fp 0($sp)
jr $ra
```

NT(e) is a compile-time constant

This question can have some other strategies, any viable strategy described clearly about how it uses temporaries and accesses activation records will get full marks. Partial marking will depend on how clearly your answer describes the strategy and how much sound that strategy is.

3. Multiple Inheritance : Consider the following class hierarchy. What would be a reasonable code generation strategy (something that is simple and performant) for a hierarchy like this. Hint: clearly specify if you are assuming virtual or replicated inheritance. [3.5]

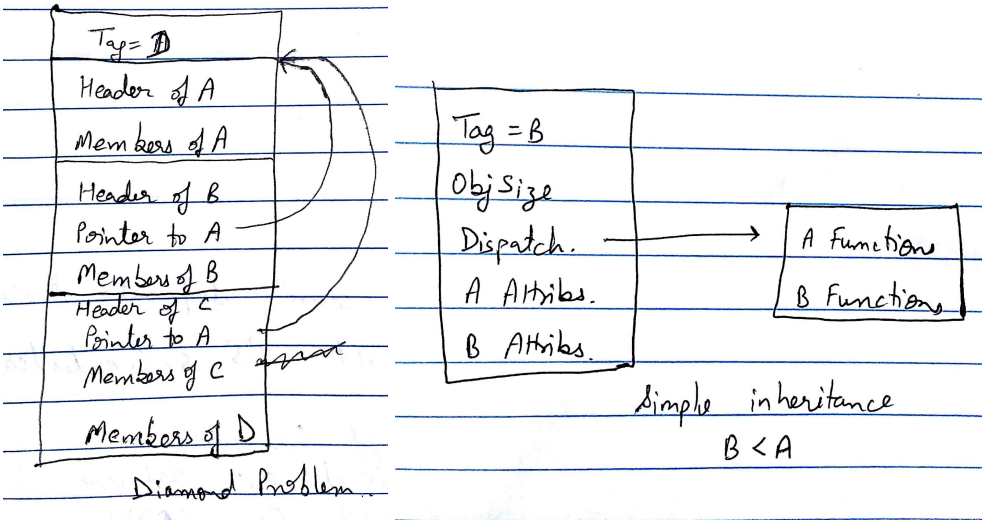
Class A { …. }

Class B inherits A { … }
Class C inherits A { … }
Class D inherits B inherits C { …. }

This is an open ended question, it can have multiple answers like the one written below. If technique is described properly and will successfully produce code for required inheritance then full marks will be given. There will be partial marks for describing strategies which will partially solve the diamond problem, or do not explain how casting an object of derived class to parent class will work (as that is also the part of semantics of inheritance).

One method of generating code for such hierarchy is by implementing objects of classes as maps where an object will be oriented as a key value pair with attribute as keys and their memory addresses as values of where they are stored. This method will be time efficient in scenarios including object casting. But it consumes a lot of unnecessary space and in some cases accessing attribute keys can also consume more time due to collisions in hash function. This is NOT an acceptable answer.

Another way is to store object by having a class type/header as first field in memory then attributes of highest parent class in hierarchy with a dispatch pointer to its function calls (e.g. class A in this case), then attributes of parent of top level class with dispatch pointer and so on and when a diamond problem occurs as in this case, then class type for object of class D will be D and then there will be attributes and dispatch pointer of top level class then its child class and so on, but headers of all the classes which inherit top class (e.g. class B & C in this case) will contain a pointer to header of class A's attributes/header as shown in figure and these attributes will have only one copy.



Both these object orientations are for case in which virtual inheritance is used. If replicated inheritance is used then those pointers to parent class in header of class B and C are not needed, both will have A's attributes and header, so class D will have two copies of A's attributes. In a language like C++, the programmer specifies the nature of inheritance

(virtual or replicated). Virtual inheritance has higher runtime cost due to an extra pointer dereference.

4. List at least two applications of formally-specified operational semantics for a programming language. Be as clear and precise as possible. [1.5]

What are operational semantics

1. Operational semantics provide independence of language execution semantics from machine architecture, instruction set, stack orientation etc. which would be the case if execution semantics are described by generated code on some machine architecture (like stack machine).
2. This independence also removes redundant descriptions, thus providing programmers some flexibility in the way they generate code for some language instruction and allows machine dependent and machine independent optimizations.

Some example applications

3. Verifying the soundness of static type systems.
4. Verifying the correctness of code generation strategy.

Vague answers like “They make understanding language execution semantics easier” are not acceptable.

Any two applications will yield full marks, only one application gets 1 or .5 mark depending on nature of application described.