CMPE 320 PRINCIPLES OF PROGRAMMING LANGUAGES
FINAL ANSWERS

1. a) Consider the following program, where the goal asks for the child of the person *father5*:

Facts and rules:

child (child1)	parent (child1, father1)
child (child2)	parent (child1, mother1)
child (child3)	parent (child2, father2)
child (child4)	parent (child2, mother2)
child (child5)	…
child (child6)	parent (child5, father5)
child (child7)	parent (child5, mother5)
child (child8)	…
child (child9)	parent (child10, father10)
child (child10)	parent (child10, mother10)

Goal:
child (X), parent (X, father5)

The first subgoal will be satisfied with the substitution *X=child1* but the second subgoal will fail and the program backtracks. There will be several backtrackings until the first subgoal matchs with *child (child5)*. However, if the two subgoals in the goal are interchanged, then first *parent (X, father5)* will be satisfied with the substitution *X=child5* and then *child (child5)* will also be satisfied. There will be no backtrackings and the program will execute much faster.

The reason of the effect of ordering propositions in this example is that there may be several children in the database but a child may have only two parents. So, it is wiser determining the son/daughter of a parent first than considering each child one by one.

b) Consider the following program for determining the ancestor relationships, where the fact indicates that a person is ancestor of himself/herself, the rule indicates that X is Y's ancestor if there is a Z such that X is Z's parent and Z is Y's ancestor, and the goal asks whether *ali* is ancestor of *ayse*:

Facts and rules:
ancestor (X,X).
ancestor (X,Y) :- ancestor (Z,Y), parent (X,Z).
Goal:
ancestor (ali,ayse).

The goal matchs with the left-hand side of the rule with the substitutions *X=ali*, *Y=ayse*. Then we have *ancestor (Z,ayse)* to be satisfied. It matchs with the fact with the substitution *Z=ayse*, but suppose that *ali* is not parent of *ayse* so *parent* proposition fails. We return to *ancestor (Z,ayse)* again and it matchs with the left-hand side of the rule with the substitution Y=*ayse*. Then we have *ancestor (Z,ayse)* again to be satisfied. This process repeats infinitely and the program does not terminate.

If the two propositions on the right-hand side of the rule are interchanged, then the program will execute correctly. In this case, first a child of *ali* (say, *ahmet*) will be found and then the program will try to determine whether *ahmet* is ancestor of *ayse*. Then, a child of *ahmet* will be found and the process will continue until we reach either to the parent of *ayse* or to a person who has no children.

2.

- Static scoping is more efficient.

  The declaration of an identifier can be determined at compile time and appropriate code can be generated. It is not necessary to search for a declaration during execution.

- Static scoping allows simplifications in machine code.

  Consider the following program:

```
main program
constant max=100
          ...
          subprogram sub1
          var. max
               …
          end
          subprogram sub2
               …
               if max>100 then
               …
               endif
               …
          end
end
```

Suppose that in this language values of constants cannot be changed within the program. If this is a statistically-scoped language and if the identifier *max* in the subprogram *sub2* refers to the constant (this is determined by the compiler), then the if construct can be completely ignored and no machine code is generated, since it will always evaluate to false during execution. On the other hand, if this is a dynamically-scoped language, such an optimization is not possible. The identifier *max* in the subprogram *sub2* may refer to the constant *max*, to the variable *max* in the subprogram *sub1*, or to another identifier *max* defined elsewhere. This depends on the calling sequence during run-time.

- Static scoping is more reliable.

  In static languages, type checking can be done during compilation. In dynamic languages, since we cannot determine the declaration of an identifier until run time, static type checking is not possible.

  Consider the following program:

```
subprogram sub1
int x
          ...
end
subprogram sub2
float x
          ...
end
subprogram sub3
          ...
          x=10.5
          ...
end
```

  If this is a static language, the compiler knows to which declaration the variable *x* in the subprogram *sub3* refers, and thus can check whether there is a type mismatch or not in the statement. However, if this is a dynamic language, the variable *x* in the subprogram *sub3* may refer to that inside *sub1* or *sub2*. This can only be known during execution, depending on the calling sequence.

- Static scoping increases readibility.

  It is usually quite difficult to understand a program code in a dynamic scoping language. The reason is the same as before; it is not easy to visualize the calling sequence and thus the references among the identifiers.

- Dynamic scoping eliminates some of parameter passings.

  When a subprogram calls another subprogram, usually the caller needs to pass some parameters to the called subprogram for communication. This is not usually necessary in dynamic scoping languages, since the identifiers of the caller are already available to the callee. Dynamic scoping is based on the idea that calling sequence in a program actually corresponds to the logic of flow of the program.

  Of course, this does not mean that there is no parameter passing concept in dynamic scoping languages. In addition to this “automatic parameter passing mechanism”, we sometimes need to pass parameters explicitly between the caller and the callee.

- In dynamic scoping, all local variables of a subprogram are visible to all the called subprograms.

  In a calling sequence like *sub1*, *sub2*, *sub3*, … (i.e. *sub1* calls *sub2*, *sub2* calls *sub3*, and so on), all local variables of *sub1* can be referenced in *sub2*, *sub3*, …. Likewise, all locals of *sub2* are visible in *sub3* and other

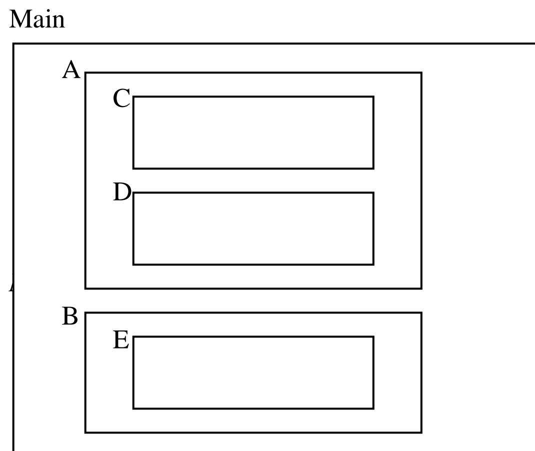
called subprograms. This is not usually the desired situation; local variables of *sub1* should be available to only *sub1*, local variables of *sub2* should be available to only *sub2*, etc. In a program, the calling sequence can give the logic of flow of the program, but not the usage pattern of the locals.

- If we think name-declaration binding not only for variables, but for all identifiers, the inefficiency of dynamic scoping becomes more clear.

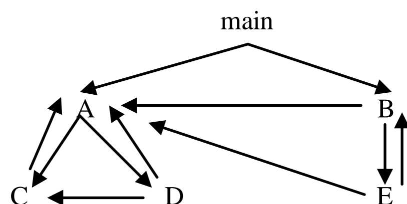
  In a program, we have several types of identifiers like variables, constants, types, subprograms, labels, etc. For each of these, the name of the identifier referenced in a statement needs to be bound to the corresponding declaration. Since all of those bindings are done during run-time in a dynamic language, there is lots of overhead to the run-time system which slows down the execution.

- Static scoping allows data sharing and subprogram calls more than necessary.

  Consider the following contour diagram:



Suppose that the scoping rules for variables and subprograms are the same as we have discussed during the lectures. Then, the possible callings in this program can be shown with the following graph:



In this graph, probably there are callings more than required. For example, we may not need that B calls A, or E calls A or B. Also, probably there are variable sharings more than required. For example, all subprograms can use all global variables.

These problems result from nested block structure. To restrict the sharings, we may change places of some of the subprograms in the code, but this does not solve all problems; it may give rise to new unwanted sharings, it may disturb the intended design, it tends to a flat block structure.

3.

- Pass-by-name is less efficient than other mechanisms.

  During implementation, it requires a parameterless subprogram call and execution, instead of just a direct of indirect addressing as in other methods.

- Readability and writability of programs may decrease.

  It is more difficult to understand the semantics of a program when reading it. It may be more confusing to write a program using pass-by-name parameters.

- This mechanism is very powerfull in some cases.

  Consider the following subprogram which computes $x = \sum_{i=1}^{n} a[i]$:

```
subprogram sum (k,low,up,value)
        s=0
        for k=low to up
            s = s+value
        end
        return (s)
end
```

We call with the statement *x=sum(i,1,n,a[i])*. If we substitute the actual parameters in place of the formal parameters, we can easily see that the subprogram gives us the desired output. In fact, this subprogram is very general. We can use the same subprogram to calculate $x=\sum_{i=1}^{m}\sum_{j=1}^{n}a[i,j]$ by calling as *x=sum(i,1,m,sum(j,1,n,a[i,j]))*. This flexibility is not provided in other parameter mechanisms. It is more difficult to do the same thing using other mechanisms.

- Some simple tasks are impossible to do using name parameters.

  Consider the following classical program which swaps two pieces of data:

```
subprogram swap (x,y)
        t=x
        x=y
        y=t
end
```

  If we call it with *call swap (i,a[i])*, as we can easily see, it does not work. It has been proved that it is impossible to write a subprogram for swaping using only name parameters. So, if pass-by-name is used in a language, then another mechanism(s) must also be incorporated; otherwise some simple subprograms cannot be written.

- Pass-by-name is more flexible.

  This flexibility comes from the late binding of the actual parameters to values. Doing something as late as possible always gives more flexibility to a language, but increases implementation cost and decreases flexibility (e.g. dynamic typing).

4. Compilation:

- call sub2(10)

  The compiler generates the necessary machine code to create the ARI of *sub2* on top of the run-time stack. The parameter will be placed into the appropriate position in the stack according to the parameter passing mechanism. The dynamic link in the ARI of *sub2* will point to the current stack pointer (SP) and then SP will be incremented to point to the top of ARI of *sub2*. To arrange the static link, the nesting depth between *Main* and the static parent of *sub2* (which is *sub1*) will be calculated. It evaluates to -1, which means that there is an error in the call: *Main* cannot call *sub2*. According to the static scoping rule for subprograms, it is not possible for a subprogram to call its grandchild. Because, at any time during execution, all static ancestors of the executing subprogram must already be in the run-time stack. However, in this example, when *sub2* begins execution, its static parent *sub1* will not be in the stack. Thus, the compiler issues a compile-time error. If there would be no error, then beginning from the static link of the caller (*Main*), the static chain would be followed nesting-depth times in order to arrange the static link of the callee (*sub2*).

- call sub3

  The process is similar to the above case. Now, the nesting depth between *sub2* and the static parent of *sub3* evaluates to 1. So, beginning from the static link of *sub2*, the static chain will be followed once, which brings us to the static parent of *sub2* (*sub1*). Thus, the static link in ARI of *sub3* will point to ARI of *sub1*.

- x=0

  Compiler determines that the reference to *x* corresponds to the declaration of *x* in *sub1*. It calculates the nesting depth between *sub3* and *sub1*, which evaluates to 1. So, it generates code to access the correct variable during run-time: beginning from the static link in ARI of *sub3*, the static chain will be followed once (which brings us to the bottom of ARI of static parent of *sub3* (*sub1*)) and by adding an offset the content of the variable *x* will be retrieved in the ARI of *sub1*.

Note that the compiler generates machine code for all of these operations. It is the run-time system that executes this machine code and performs the desired operations.

Execution:

- call sub2(10)

  The run-time system executes the code generated by the compiler. During execution, the ARI of *sub2* is created on the stack, the fields in the ARI (parameters, local variables, return status, return value, static and dynamic links, etc.) are given values. The static and dynamic links are arranged as explained above. After the ARI is created, the stack pointer is incremented and the execution jumps to the first instruction in the code segment of *sub2*.

- call sub3

  The process is similar to the above case. ARI of *sub3* is created and execution continues from the code segment of *sub3*.

- x=0

  As indicated by the generated code, beginning from the static link in ARI of *sub3*, the system follows the static chain once. Then, an offset is added to the address of the bottom of the ARI where the variable resides and the variable is retrieved.