Quiz Review

Q1. What is the advantage of binding things as early as possible? Is there any advantage to delaying binding?

Answer:

Early binding generally leads to greater efficiency (compilation approach)

Late binding general leads to greater flexibility

Q2. For the following grammar:
S →a S b | c E
E →d | f E a

(1) Circle the strings that can be generated by the grammar and cross out those that cannot:
cfda (T) acfab (F) aacdbb (T) aacfdab (F)

(2) The grammar is ambiguous. True or false (circle one)
False

Q3.

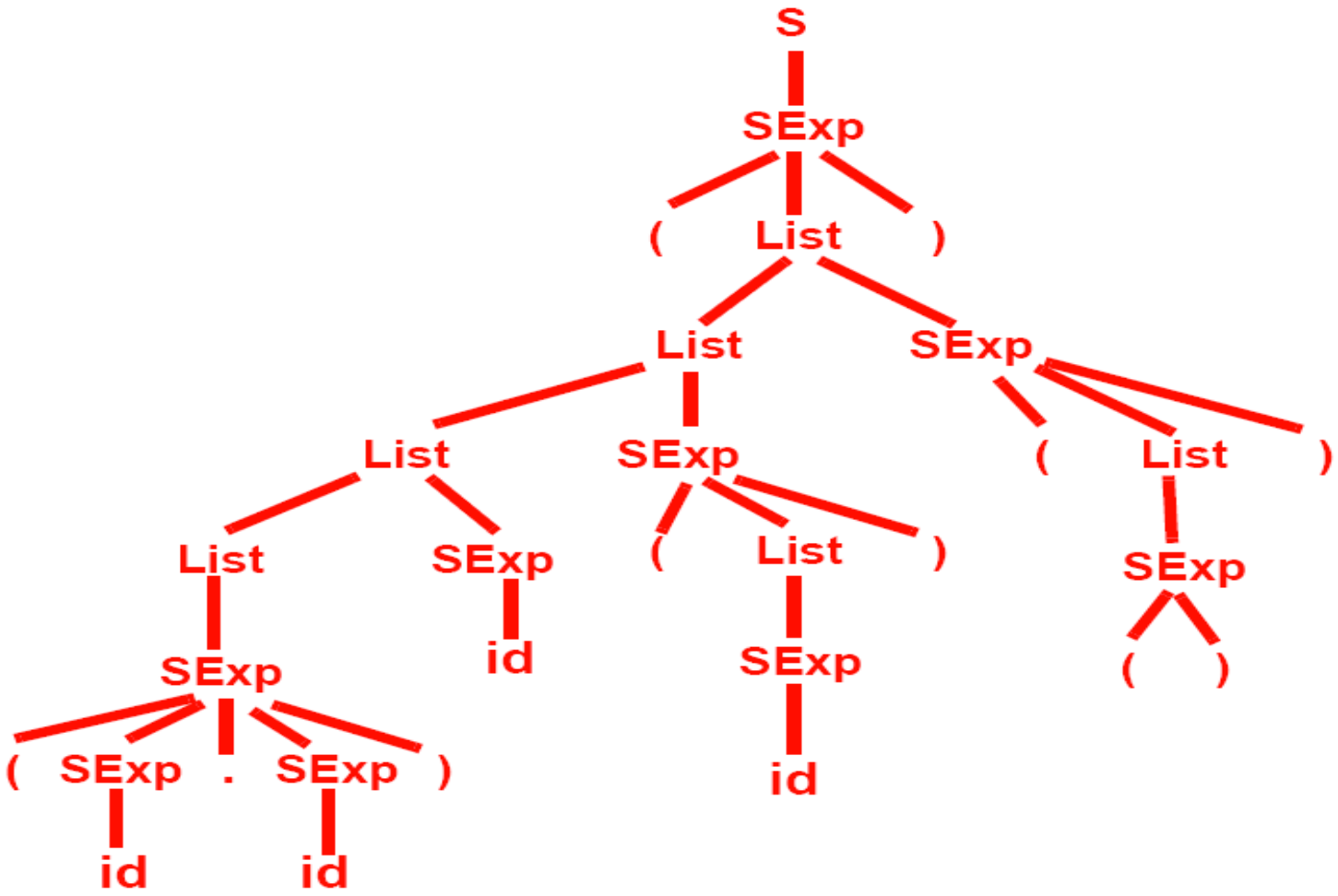
Grammar G

1.	S	→	SExp
2.	SExp	→	id
3.		|	(SExp . SExp)
4.		|	(List)
5.		|	( )
6.	List	→	List SExp
7.		|	SExp

a. Using grammar G, draw a parse tree (also known as a derivation tree) for the following sentence:

```
( ( a . b ) c ( d ) ( ( ) ) )
```

In this parse tree, the root node is labelled S; leaf nodes are labelled with terminal symbols.


S
SExp
(List)
List
SExp
List
SExp
(List)
List
SExp
(List)
SExp
SExp
id
( )
SExp
(SExp . SExp)
id
id
id

b. Write down a rightmost derivation for the sentence given in part (a).
Hints: In a derivation, the first line is the goal symbol and the last line is the sentence; the derivation is rightmost if the rightmost nonterminal symbol is expanded at each step.

S
SExp
(List)
(List SExp)
(List (List))
(List (SExp))
(List (( )))
(List SExp (( )))
(List (List) (( )))
(List (SExp) (( )))
(List (id) (( )))
(List SExp (id) (( )))
(List id (id) (( )))
(SExp id (id) (( )))
((SExp . SExp) id (id) (( )))
((SExp . id) id (id) (( )))
((id . id) id (id) (( )))

Note: no standard parsing method performs a rightmost canonical derivation.

LL(1) or Recursive Descent perform a leftmost canonical *derivation*;

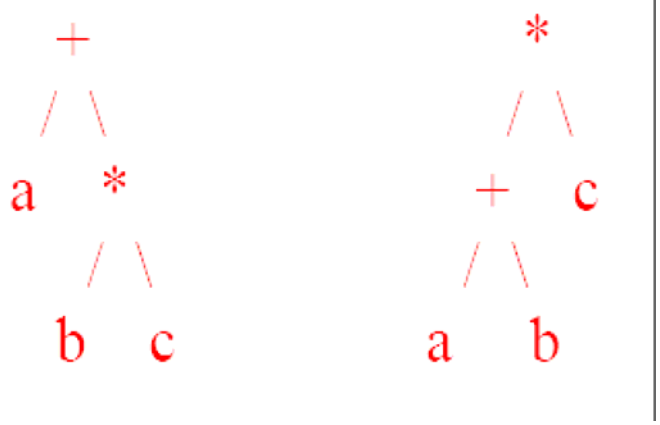
The bottom-up methods perform a rightmost canonical *parse*, which is the same as shown here but where one reads the lines from the bottom to the top.

Q4. Do a top down leftmost derivation of the following string given the grammar listed below:

a = b * (c + a)

```
S → ID = E
ID → a | b | c
E → E + E
  | E * E
  | ( E)
  | ID
```

```
S  →  ID = E
   →  a = E
   →  a = E * E
   →  a = ID * E
   →  a = b * E
   →  a = b * (E)
   →  a = b * (E + E)
   →  a = b * (ID + E)
   →  a = b * (c + E)
   →  a = b * (c + ID)
   →  a = b * (c + a)
```



Is the grammar from above ambiguous or not? Why or why not?

Yes. An expression such as: a + b * c can generate two different parse trees.

Q5. LR parser

How a string like "1 + 1" would be parsed by LR parser.

Grammar:

(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1

	action						*goto*	
state	*	+	0	1	$		E	B
0			s1	s2			3	4
1	r4	r4	r4	r4	r4			
2	r5	r5	r5	r5	r5			
3	s5	s6			acc			
4	r3	r3	r3	r3	r3			
5			s1	s2				7
6			s1	s2				8
7	r1	r1	r1	r1	r1			
8	r2	r2	r2	r2	r2			

Stack	*Input*	*Action*
0	1+1$	Shift 2
0‘1’2	+ 1 $	Reduce 5 (Use GOTO[0, B])
0‘B’4	+ 1 $	Reduce 3 (Use GOTO[0, E])
0‘E’3	+ 1 $	Shift 6
0‘E’3‘+’6	1 $	Shift 2
0‘E’3‘+’6‘1’2	$	Reduce 5 (Use GOTO[6, B])
0‘E’3‘+’6‘B’8	$	Reduce 2 (Use GOTO[0, E])
0E3	$	Accept

Type Checking

The process of checking that for each operation, the types of its operands are appropriate, i.e. the operation is legal for these types.

In all programming languages, every value has a certain type, and the compiler and/or the runtime system knows what the type is.

Static versus Dynamic Type-Checking

***Static* type-checking:** type-checking done by the compiler, before executing the program.

- int i;
  int j;
  i = j;

Java and C# are Strongly Typed = the compiler checks that every assignment and every method call is type correct.

```
public class Num {
// class providing useful numeric routines
   public static int gcd (int n, int d) {
     while (n != d)
          if (n > d) n = n  - d; else d = d - n;
     return n;
    }
}
```

```
int y = 7, z = 3;
int x =Num.gcd (z,y);
```

C and C++ are not strongly typed language because both include union types, which are not type checked.

• *Type checking* is the activity of ensuring that the operands of an operator are of compatible types.

• A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type.

```
int i;
double c = i;

Object o1 = new Car();
```

– This automatic conversion is called a **coercion**.

• A *type error* is the application of an operator to an operand of an inappropriate type

The advantages of static type-checking

- Potential errors can be identified earlier.
- More care is needed in the program design and implementations can take advantage of the additional information to produce more efficient programs with less runtime checking code.

The penalty of static checking is reduced programmer flexibility.

***Dynamic* type-checking:** type-checking done while executing the program.

```
Shape s = new Circle();
Shape s = new Triangle();
Shape s = new Line();
```

```
Shape s = new Circle();
      s.draw();
```

```
Shape s = new Line();
      s.draw();
```

- Dynamic type checking gives more freedom and flexibility to the programmer.
- At run-time, type-checking can know the actual values of variables, not just their initial value or declared type.
- This flexibilty is at the cost that type checking errors now occur unpredictably at run-time.

- All languages combine run-time and compile-time type-checking (Java, etc.)
- Some languages, such as JavaScript and PHP, allow only dynamic type checking.

Scope: The range of statements over which it is visible

Local variable

Declared in the same program subunit (program, subprogram, function, etc.) in which it is used.

Nonlocal variable (static-scoped language. Such as Ada.)

A variable not declared in the same program subunit in which it is used, but not available to every subunit in the program.

Global variable

A variable not declared in the same program subunit in which it is used, and available to every subunit in the program.

```
void foo (int x) {  // begin scope of x
   int y;       // begin scope of y
   ...
   y=0;  }            // end scope of x and y
```

In most block-structured languages, the scope of a name is its enclosing block.

```
void foo (int x) {  // begin scope of x
  if (x == 0) {     // begin scope of y
  int y = 0;
  ...
   }                // end scope of y
   ...
   for (int z=0;    // begin scope of z
    z<5; z++) {
  ...
   }                // end scope of z
   ...
 }                  // end scope of x
```

Static Scope versus Dynamic Scope

Static Scope: the scope of a variable can be statically determined prior to execution. Called static scoping. (Introduced by ALGOL 60)

Two categories of Static-scoped language:

(1) Subprograms can be nested, which created nested static scope, (Ada, JavaScript, and PHP)

(2) Subprograms cannot be nested (C-based languages).

(3) We focus on languages that allow nested subprograms.

To connect a name reference to a variable, you must find the declaration.

Consider the following Ada procedure

```
procedure Big is
   X:   Integer;
   procedure Sub1 is
     begin
        ….. X …..
     end;
   procedure Sub2 is
     X : Integer;
     Begin
      ……..
     end;
  begin    --- of Big
     ….
   End      --- of Big
```

- Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Enclosing static scopes (to a specific scope) are called its static ancestors; the nearest static ancestor is called a static parent

Static Scope is associated with the static text of program.

Can determine scope by looking at structure of program rather than execution path telling how got there.

- Variables can be hidden from a unit by having a "closer" variable with the same name
- C++ and Ada allow access to these "hidden" variables
  - In Ada: unit**.name**
    X declared in Big can be accessed in Sub2 by the reference Big.X
  - In C++: using scope operator (::)
    ::name

```
#include < iostream >
void func();
int i = 5;
int j = 3;
float f = 10.0;

int main() {
   int j = 7;
   cout << "i in main: " << i << endl;
   cout << "j in main: " << j << endl;
   cout << "global j:  " << ::j << endl;
   func();
   return 0;
}

void func() {
   float f = 20.0;
   cout << "f in func: " << f << endl;
   cout << "global f:  " << ::f << endl;
}
```

Blocks

-A method of creating static scopes inside program units--from ALGOL 60

-Examples:

C and C++:
```
for (...) {
    int index;
    ...
}
```

Ada:
```
declare LCL : FLOAT;
begin
    ...
end
```

The scope created by blocks are treated exactly like those created by subprograms. Reference to variables in a block that are not declared there are connected to declarations by searching enclosing scopes in order of increasing size.

- C++ allow variables definition to appear anywhere in function. When a definition appears at a position other than at the beginning of a function, that variable's scope is from its definition statement to the end of function.
- In C, all data declarations in a function must appear at the beginning of the function.
- The for statement of C++, Java and C# allow variable definition in their initialization expression. The scope is restricted to the for construct, as the case with Java and C#.
- The classes of C++, Java and C# treat their instance variable differently from the variable defined in their methods. The scope of a variable defined in a method start at the definition. However, regardless of where a variable is defined in a class, its scope is the whole class

Dynamic Scope

- Based on calling sequences of program units
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point

```
MAIN
    - declaration of x
      SUB1
        - declaration of x -
        ...
        call SUB2
        ...
      SUB2
        ...
        - reference to x -
        ...
    ...
    call SUB1
...
```

MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

- Dynamic Scope is associated with the execution path of program.
- LISP and APL use dynamic scoping

Scope Example

- Static scoping
  - Reference to x of SUB2 is to MAIN's x
- Dynamic scoping
  - Reference to x of SUB2 is to SUB1's x

- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - Disadvantage: poor readability

```
program ...
    var A : integer;
    procedure Y(...);
        begin
            ...; B := A + B; ...
        end; {Y}
    procedure Z(...);
        var A: integer;
        begin
            ...; Y(...); ...
        end; {Z}
    begin {main}
        ...; Z(...);...
    end.
```

Question: Which variable with name A is used when Y is called from Z?

- In static, clearly globally defined A.
- In dynamic, local A in Z (since declaration of A in Z is most recent).

Variable categories by lifetime and memory location:

1. static variables
2. stack-dynamic variables
3. heap-dynamic variables

Static variables

- Bound to a memory location before execution, and do not move.
- Lifetime = entire execution.
- Called `static` in Java, C, and C++.

Pros and cons:

- No overhead at runtime for allocation and deallocation.
- Can't have recursion if only have static variables.
  E.g., early Fortran.
- When finished using a variable, no other variable can use its space.

```
#include <iostream.h>

void showstat( int curr ) {
  static int nStatic=0;    // Value of nStatic is retained between each function call
  nStatic += curr;
  cout << "nStatic is " << nStatic << endl;
}

void main() {
  for ( int i = 0; i < 5; i++ )
    showstat( i );
}
```

```
Java:  public class Counter {
                public static int count=0;
                public Counter (){
                  }

                public static void counting_function(){
                     System.out.println("count = " +
               ++count);
                  }

                public static void main (String[] args) {
                     counting_function();    //   1
                     counting_function();    //   2
                     counting_function();    //   3
                     counting_function();    //   4
                     counting_function();    //   5
                 }
```

Imagine you wanted to count how many Duck instances are being created while your program is running.

Non-static variable:

```
class Duck {
      int duckCount = 0;
       public Duck() {
            duckCount++;
}
```

Static variable:

```
public class Duck {
  private int size;
  private static int duckCount = 0;
 public Duck () {
       duckCount++;
}
}
```

Programming Language Support:

C, C++, Java: include the "static" specifier on a local variable definition.

FORTRAN I, II, IV: all variables static.

Pascal: no support.

- Static variable is a class variable, rather than an instance variable.

Stack-dynamic variables

- Bound to a memory location when the variable's declaration is executed.
- Lifetime = from execution of declaration until end of the "block" in which it occurs.
- Called "stack-dynamic" because stored in an activation record, which is pushed/popped on the run-time stack when entering/leaving a block.
- This is the default for local variables in Java, C, and C++.

Pros and cons:

- Overhead at runtime for allocation and deallocation.
- Allows recursion to work properly.
- Efficient use of space because deallocate when pop.

In most current programming languages, the formal parameters and local variables of subroutines (functions, methods) are stack-dynamic variables.

```
int foo (int a, int b) {
    int x = 4;
    a = a + bar(b);
    return x + a + b;
}
```

Heap-dynamic variables

nameless variables allocated on the heap for dynamic data structures or for instances of objects in OO programming languages.

Heap-dynamic variables

- Bound to a memory location when explicit instructions are executed at runtime.
- Nameless; just referred to by a pointer.
- Lifetime = from allocation instruction to deallocation instruction.
- Called "heap-dynamic" because the memory is taken from the heap.
- C: malloc free
  C++: new delete
  Java: new

Pros and cons:

- Good for building dynamic structures.
- Hard to write the code correctly.
- A little slower to execute.

In Java or C++, heap-dynamic variables are allocated by the **new** operator.

```
Car c = new Car();
```

In C, the allocation operator is the function **malloc(size)**,

```
#include <stdlib.h>

/* Allocate space for an array with 10 elements of type int */
int *ptr = (int *) malloc ( sizeof(int)  * 10 );
if (ptr == NULL)
   exit(1); /* We could not allocate any memory, so exit */
```

Memory allocated via malloc is persistent: it will continue to exist until the program terminates or the memory is deallocated by the programmer .

```
void free(ptr)
```

• dynamic objects in C++ (via new and delete),

```
int *intnode;
. . .
intnode = new int;
. . .
delete intnode;
```

Heap-Dynamic Variables

Advantage: high flexibility, convenient to implement data structures, such as liked list and trees.

Disadvantage: (1) difficult to use pointer and reference correctly.
(2) low reliability, complexity of storage management implementation.

- Solutions in Java:
  - remove pointers (only use references)
  - array bound checking
  - garbage collection

Referencing Environments

- *referencing environment* of a statement is the collection of all names that are visible in the statement
- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes

```
procedure Example is
   A, B, : Integer;
   ...

   procedure Sub1 is
      X, Y : Integer;
      begin -- Sub1
      ...          <============ 1
      end;

   procedure Sub2 is
      X : Integer;
      ...

      procedure Sub3 is
         X : Integer;
         begin -- Sub3
         ...      <============ 2
         end;
```

```
      begin -- Sub2
      ...          <============ 3
      end;

   begin -- Example
   ...             <============ 4
   endl;

Point     Referencing Environment
   1
   2
   3
   4
```

- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms
- A subprogram is active if its execution has begun but has not yet terminated

```
void sub1() {
   int a, b;
    ……        <-------- 1
}   / *  end of sub1   */
void sub2() {
   int b, c;
     ……       <-------- 2
   sub1;
} /* end of sub2   */
void main() {
   int c, d;
   …….        <-------- 3
   sub2();
 } /* end of main   */
```

The reference environments of the indicated program points are as follows:

Point	Referencing Environment
1.	a and b of sub1, c of sub2, d of main, (c of main and b of sub2 are hidden)
2.	b and c of sub2, d of main, (c of main is hidden)
3.	c and d of main

Named Constants

- a variable that is bound to a value only once when it is bound to storage

Advantages:

- readability can be improved, for example, by using pi instead of 3.14159
- Used to parameterize programs

```
void example(){
int [] intList = new int[100];
….
for (index = 0; index<100; index++){
……
}

for (index = 0; index<100; index++){
……
}
….
Average=sum/100;
…
}
```

```
void example(){
final int len =100;
int [] intList = new int[len];
….
for (index = 0; index<len; index++){
……
}

for (index = 0; index<len; index++){
……
}
….
Average=sum/len;
…
}
```