Solved Past Paper TPL Finals Fall 2018

Q1a. Explain static and stack dynamic variables.

Chapter 5: Slide 20 – 21

Categories of Variables by Lifetimes

- Static--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution, e.g., C and C++ `static` variables in functions
  - Advantages: efficiency (direct addressing), history-sensitive subprogram support
  - Disadvantage: lack of flexibility (no recursion)

20

Categories of Variables by Lifetimes

- Stack-dynamic--Storage bindings are created for variables when their declaration statements are *elaborated*.
  (A declaration is elaborated when the executable code associated with it is executed)
- If scalar, all attributes except address are statically bound
  - local variables in C subprograms (not declared **`static`**) and Java methods
- Advantage: allows recursion; conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

21

Q1b. What are scope and life time of a variable?

Chapter 5: Slide 39

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a **`static`** variable in a C or C++ function

Book: Page No 246 – 247 (ager slide confusing lgay tau read below script from book)

5.6 Scope and Lifetime

Sometimes the scope and lifetime of a variable appear to be related. For example, consider a variable that is declared in a Java method that contains no method calls. The scope of such a variable is from its declaration to the end of the method. The lifetime of that variable is the period of time beginning when the method is entered and ending when execution of the method terminates. Although the scope and lifetime of the variable are clearly not the same, because static scope is a textual, or spatial, concept whereas lifetime is a temporal concept, they at least appear to be related in this case.

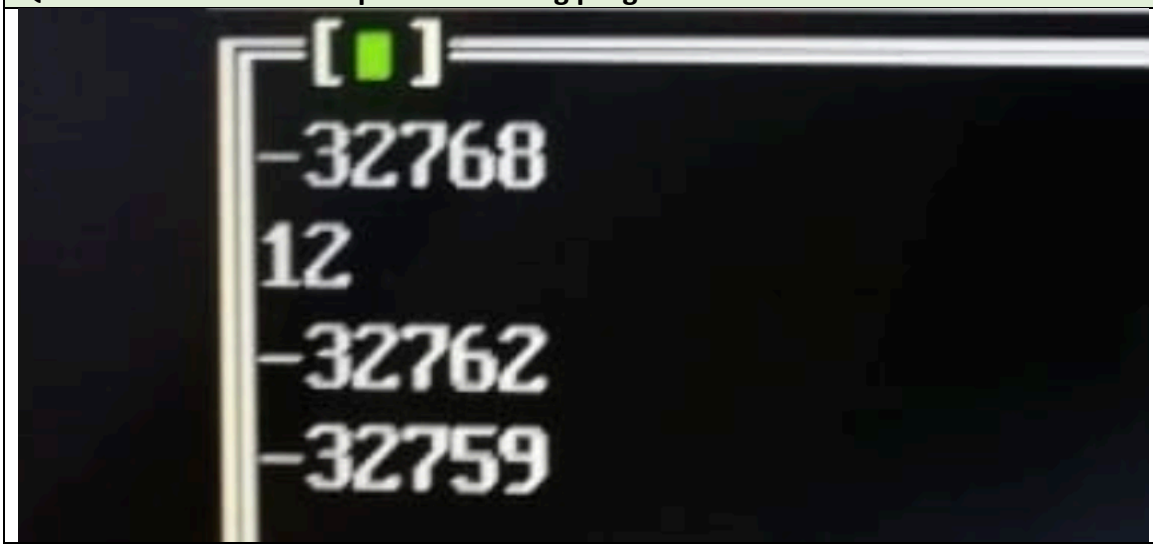
This apparent relationship between scope and lifetime does not hold in other situations. In C and C++, for example, a variable that is declared in a function using the specifier **`static`** is statically bound to the scope of that function and is also statically bound to storage. So, its scope is static and local to the function, but its lifetime extends over the entire execution of the program of which it is a part.

Scope and lifetime are also unrelated when subprogram calls are involved. Consider the following C++ functions:

```
void printheader() {
  . . .
 }  /* end of printheader */
void compute() {
  int sum;
  . . .
  printheader();
 } /* end of compute */
```

The scope of the variable `sum` is completely contained within the `compute` function. It does not extend to the body of the function `printheader`, although `printheader` executes in the midst of the execution of `compute`. However, the

Q1c. What will be the output of following program?



Q2a. Discuss homogenous and heterogenous arrays.

Homogenous arrays
Chapter 6. Slide 21,

Array Types

- An array is a homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element.

Heterogenous arrays
Chapter 6: Slide 30

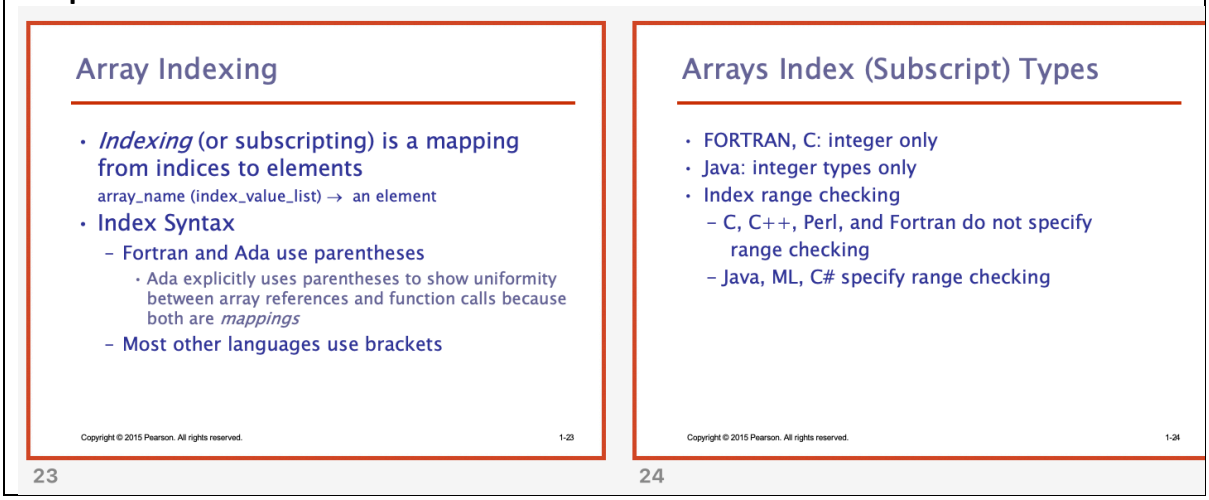
Heterogeneous Arrays

- A *heterogeneous array* is one in which the elements need not be of the same type
- Supported by Perl, Python, JavaScript, and Ruby

Q2b. A new programming language MyPL is designed by one of your friends. Do we agree with his design choices?

Q2b Part a: Array subscripts can be an integer or an integer expression. Subscript cannot be of any other data type including enum data types.

Chapter 6: Slide 23 – 24



Array Indexing

- *Indexing* (or subscripting) is a mapping from indices to elements
  array_name (index_value_list) → an element
- Index Syntax
  - Fortran and Ada use parentheses
    - Ada explicitly uses parentheses to show uniformity between array references and function calls because both are *mappings*
  - Most other languages use brackets

23

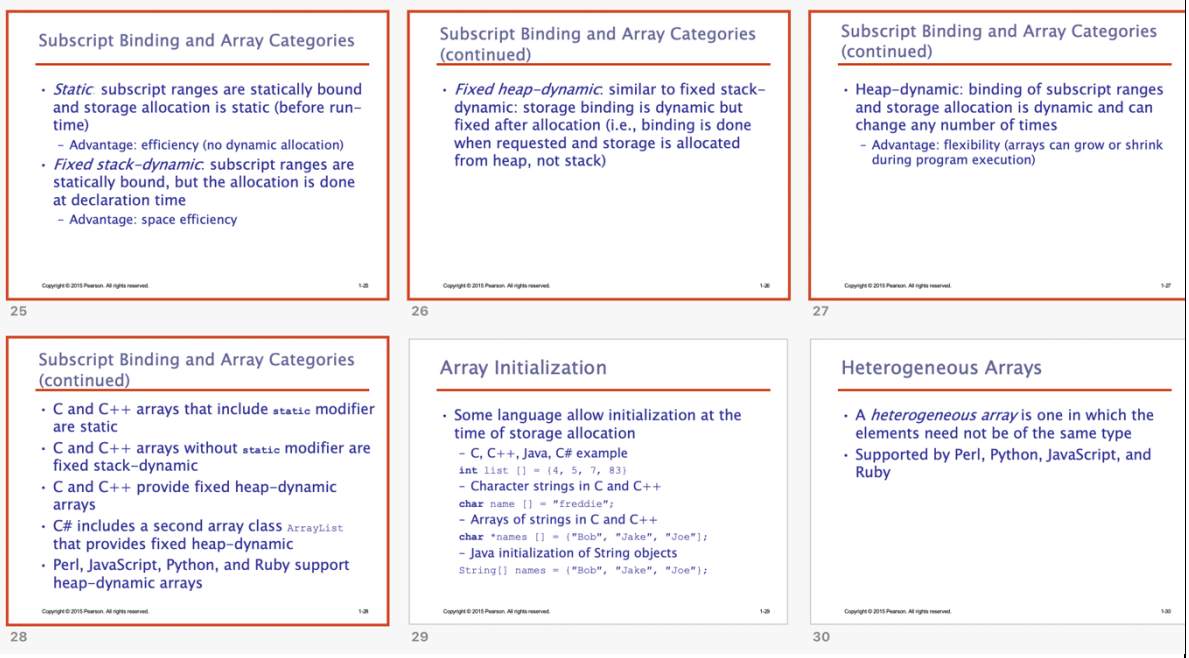
Arrays Index (Subscript) Types

- FORTRAN, C: integer only
- Java: integer types only
- Index range checking
  - C, C++, Perl, and Fortran do not specify range checking
  - Java, ML, C# specify range checking

24

Q2b Part b: Subscript range checking is not done.

Chapter 6: Slide 25 – 28



Subscript Binding and Array Categories

- *Static*: subscript ranges are statically bound and storage allocation is static (before run-time)
  - Advantage: efficiency (no dynamic allocation)
- *Fixed stack-dynamic*: subscript ranges are statically bound, but the allocation is done at declaration time
  - Advantage: space efficiency

 1-25

25

Subscript Binding and Array Categories (continued)

- *Fixed heap-dynamic*: similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation (i.e., binding is done when requested and storage is allocated from heap, not stack)

 1-26

26

Subscript Binding and Array Categories (continued)

- Heap-dynamic: binding of subscript ranges and storage allocation is dynamic and can change any number of times
  - Advantage: flexibility (arrays can grow or shrink during program execution)

 1-27

27

Subscript Binding and Array Categories (continued)

- C and C++ arrays that include `static` modifier are static
- C and C++ arrays without `static` modifier are fixed stack-dynamic
- C and C++ provide fixed heap-dynamic arrays
- C# includes a second array class `ArrayList` that provides fixed heap-dynamic
- Perl, JavaScript, Python, and Ruby support heap-dynamic arrays

 1-28

28

Array Initialization

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, C# example

```
int list [] = {4, 5, 7, 83}
```

  - Character strings in C and C++

```
char name [] = "freddie";
```

  - Arrays of strings in C and C++

```
char *names [] = {"Bob", "Jake", "Joe"};
```

  - Java initialization of String objects

```
String[] names = {"Bob", "Jake", "Joe"};
```

 1-29

29

Heterogeneous Arrays

- A *heterogeneous array* is one in which the elements need not be of the same type
- Supported by Perl, Python, JavaScript, and Ruby

 1-30

30

Book: Page 276, Ager slides confusing lgay tau read below from book.

Subscript Bindings and Array Categories

The binding of the subscript type to an array variable is usually static, but the subscript value ranges are sometimes dynamically bound.

In some languages, the lower bound of the subscript range is implicit. For example, in the C-based languages, the lower bound of all subscript ranges is fixed at `0`. In some other languages, the lower bounds of the subscript ranges must be specified by the programmer.

There are four categories of arrays, based on the binding to subscript ranges, the binding to storage, and from where the storage is allocated. The category names indicate the design choices of these three. In the first three of these categories, once the subscript ranges are bound and the storage is allocated, they remain fixed for the lifetime of the variable. Of course, when the subscript ranges are fixed, the array cannot change size.

A **static array** is one in which the subscript ranges are statically bound and storage allocation is static (done before run time). The advantage of static arrays is efficiency: No dynamic allocation or deallocation is required. The disadvantage is that the storage for the array is fixed for the entire execution time of the program.

A **fixed stack-dynamic array** is one in which the subscript ranges are statically bound, but the allocation is done at declaration elaboration time during execution. The advantage of fixed stack-dynamic arrays over static arrays is space efficiency. A large array in one subprogram can use the same space as a large array in a different subprogram, as long as both subprograms are not active at the same time. The same is true if the two arrays are in different blocks that are not active at the same time. The disadvantage is the required allocation and deallocation time.

A **fixed heap-dynamic array** is similar to a fixed stack-dynamic array, in that the subscript ranges and the storage binding are both fixed after storage is allocated. The differences are that both the subscript ranges and storage bindings are done when the user program requests them during execution, and the storage is allocated from the heap, rather than the stack. The advantage of fixed heap-dynamic arrays is flexibility—the array's size always fits the problem. The disadvantage is allocation time from the heap, which is longer than allocation time from the stack.

A **heap-dynamic array** is one in which the binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array's lifetime. The advantage of heap-dynamic arrays over the others is flexibility: Arrays can grow and shrink during program execution as the need for space changes. The disadvantage is that allocation and deallocation take longer and may happen many times during execution of the program. Examples of the four categories are given in the following paragraphs.

Q2b Part C: Array initialization is not allowed.

Chapter 6: Slide 29

Array Initialization

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, C# example

```
int list [] = {4, 5, 7, 83}
```

  - Character strings in C and C++

```
char name [] = "freddie";
```

  - Arrays of strings in C and C++

```
char *names [] = {"Bob", "Jake", "Joe"];
```

  - Java initialization of String objects

```
String[] names = {"Bob", "Jake", "Joe"};
```

Q3a. What is the difference between pretest and posttest loops? Discuss pretest and posttest loops of C/C++.

Book: Chapter 8 - Page 373

statements. The pretest and posttest logical loops have the following forms:

```
while (control_expression)
  loop body
```

and

```
do
  loop body
while (control_expression);
```

These two statement forms are exemplified by the following C# code segments:

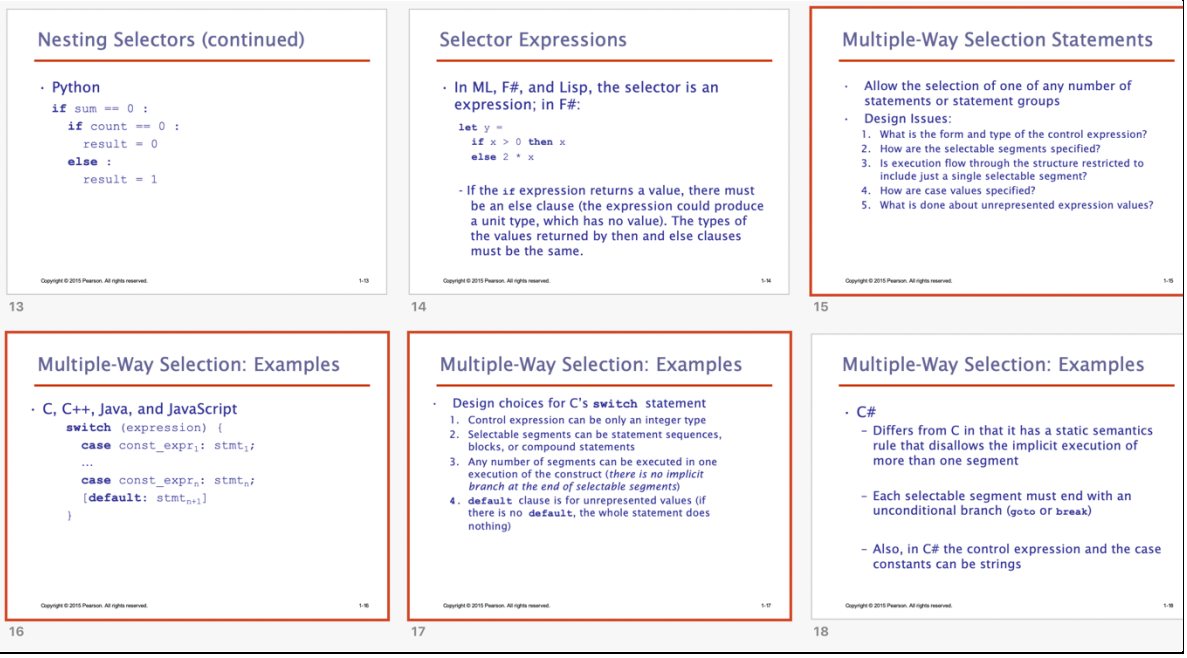
```
sum = 0;
indat = Int32.Parse(Console.ReadLine());
while (indat >= 0) {
  sum += indat;
  indat = Int32.Parse(Console.ReadLine());
}

value = Int32.Parse(Console.ReadLine());
do {
   value /= 10;
   digits ++;
}  while  (value > 0);
```

Note that all variables in these examples are integer type. The `ReadLine` method of the `Console` object gets a line of text from the keyboard. `Int32.Parse` finds the number in its string parameter, converts it to **`int`** type, and returns it.

In the pretest version of a logical loop (**`while`**), the statement or statement segment is executed as long as the expression evaluates to true. In the posttest version (**`do`**), the loop body is executed until the expression evaluates to false. In both cases, the statement can be compound. The operational semantics descriptions of those two statements follow:

Q3b. Design issues in multiple selection constructs. Such as switch statement in C/C++. Elaborate each of them.

Chapter 8: Slide 15 – 17



Nesting Selectors (continued)

- Python

```
if sum == 0 :
  if count == 0 :
    result = 0
  else :
    result = 1
```

 1-13

13

Selector Expressions

- In ML, F#, and Lisp, the selector is an expression; in F#:

```
let y =
  if x > 0 then x
  else 2 * x
```

- If the `if` expression returns a value, there must be an else clause (the expression could produce a unit type, which has no value). The types of the values returned by then and else clauses must be the same.

 1-14

14

Multiple-Way Selection Statements

- Allow the selection of one of any number of statements or statement groups
- Design Issues:
  1. What is the form and type of the control expression?
  2. How are the selectable segments specified?
  3. Is execution flow through the structure restricted to include just a single selectable segment?
  4. How are case values specified?
  5. What is done about unrepresented expression values?

 1-15

15

Multiple-Way Selection: Examples

- C, C++, Java, and JavaScript

```
switch (expression) {
  case const_expr1: stmt1;
  ...
  case const_exprn: stmtn;
  [default: stmtn+1]
}
```

 1-16

16

Multiple-Way Selection: Examples

- Design choices for C's `switch` statement
  1. Control expression can be only an integer type
  2. Selectable segments can be statement sequences, blocks, or compound statements
  3. Any number of segments can be executed in one execution of the construct (*there is no implicit branch at the end of selectable segments*)
  4. `default` clause is for unrepresented values (if there is no `default`, the whole statement does nothing)

 1-17

17

Multiple-Way Selection: Examples

- C#
  - Differs from C in that it has a static semantics rule that disallows the implicit execution of more than one segment
  - Each selectable segment must end with an unconditional branch (`goto` or `break`)
  - Also, in C# the control expression and the case constants can be strings

 1-18

18

Q4a. Convert the below BNF to EBNF.

Book: Chapter 3, Page 150

EXAMPLE 3.5 BNF and EBNF Versions of an Expression Grammar

BNF:

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
<factor> → <exp> ** <factor>
         | <exp>
<exp> → (<expr>)
      | id
```

EBNF:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → <exp> { ** <exp>}
<exp> → (<expr>)
      | id
```

Q4b. How lexical analyzer interacts with syntax analyzer?

Theoretical Portion of answer

Book: Chapter 4 - Page 188

```
result = oldsum - value / 100;
```

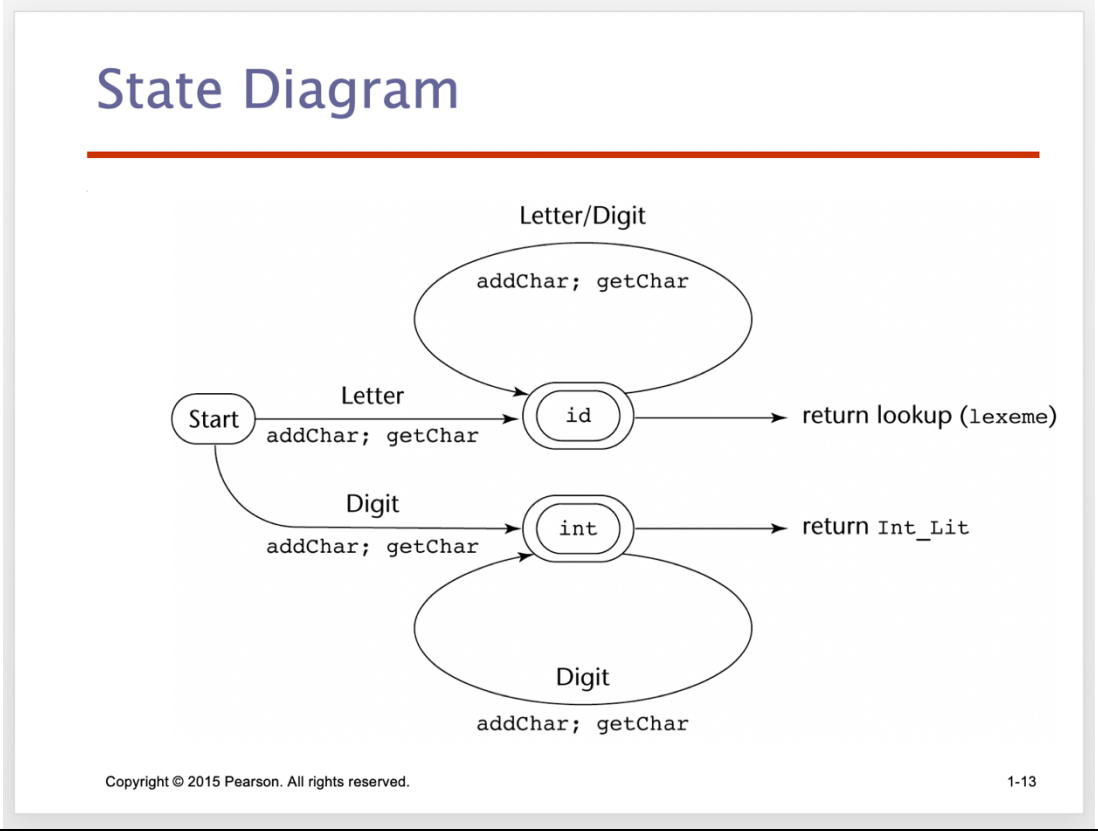
Following are the tokens and lexemes of this statement:

Token	*Lexeme*
IDENT	result
ASSIGN_OP	=
IDENT	oldsum
SUB_OP	-
IDENT	value
DIV_OP	/
INT_LIT	100
SEMICOLON	;

Lexical analyzers extract lexemes from a given input string and produce the corresponding tokens. In the early days of compilers, lexical analyzers often processed an entire source program file and produced a file of tokens and lexemes. Now, however, most lexical analyzers are subprograms that locate the next lexeme in the input, determine its associated token code, and return them to the caller, which is the syntax analyzer. So, each call to the lexical analyzer returns a single lexeme and its token. The only view of the input program seen by the syntax analyzer is the output of the lexical analyzer, one token at a time.

Graphical representation

Chapter 4: Slide 13



Q5. Create a parse tree using bottom up approach.

Book: Chapter 4 – Page 203

Figure 4.2

Parse tree for
(sum + 47) / total