POPL II A : JAN 20 : Homework

Raj Patil : CS18BTECH11039

February 7, 2020

This document was generated in LaTeX

Chapter 6: CHECK YOUR UNDERSTANDING

17>

The order of evaluation of the operands of a function (an operator is a function as well) is left unspecified due to the following:

it allows the compiler to decide the optimal order of evaluation

for instance, in the case of a∗ b + f(c) , it is better to evaluate a∗ b later as it needs to be saved in some registers which may be needed by f(c) in the future: resulting in overheads when caller/callee saving

however, the compiler needs to be explicit about the order of evaluation in the case when function exhibit side-effects as this may alter the final result.

18>

A short-circuit Boolean evaluation scheme stops further evaluation when the final value of the predicate is determined. For instance, the moment when you first encounter a True when evaluating consecutive inclusive ORs. It is wasteful to evaluate the following Booleans.

This may also be used smartly to stop the execution at a certain point if you do not wish to do so, for instance: if(!(ptr==NULL) & & f(ptr)) will stop execution the moment it finds the pointer ptr points to NULL, thus elegantly saving us from an illegal reference in the future.

12>

Initialization of a variable is beneficial in the following ways:

run-time costs are saved if a statically-allocated variable is initialized by the compiler itself.

oblivious usage of uninitialized variables is a common programming error and this can be a source of non-determinism in the execution of a program making it difficult to trace back to a point of error. Initializing all the concerned variables by default to a particular value eliminates this and allows us to reproduce errors to track them down.

2>

The term operator is traditionally used for built-in functions that use reserved, simplified syntax and have with them, the word operand associated to signify their arguments. User-defined functions on the other hand, have a specific syntax defined at design time and syntactic sugar is not a priority.

28>

Floating point numbers are not used when iterating over a procedure due to fact that floating point arithmetic is not commutative and also that their representation is most dense near 0 and gets sparser further away in most implementations. This loss of precision may lead to non-deterministic results or implementation dependent results which is not good.

16>

Checking for definite initialization of a variable by the compiler can be tedious and difficult as the usage of a variable depends on the control flow of the program and this means the compiler has to check if the concerned variable is bounded to a deterministic value at every possible control flow branch which is wasteful and some variables will be used in a select number of possibilities.

Chapter 6: EXERCISES

1>

No, these actions are not contradictory. They are in fact orthogonal to each other. Evaluating an operand and performing the concerned operation is different.

for instance:

consider the two executions:

(1∗ 2)+(2∗ 3)+(3∗ 6)

=2+(2∗ 3)+(3∗ 6)

=2+6+(3∗ 6)

=8+(3∗ 6)

=8+18

=26

and

(1∗ 2)+(2∗ 3)+(3∗ 6)

=(1∗ 2)+6+(3∗ 6)

=(1∗ 2)+6+18

=2+6+18

=8+18

=26

both are left associative (observing + for now) but the operands are evaluated in a different order.

4>

prefix notation:

/(+-(0)(b)(sqrt(-∗ (b)(b)(∗ 4∗ (a)(c)))))(∗ (2)(a))

postfix notation:

((0)(b)-)(sqrt((b)(b)∗ ((4)(a)∗ (c)∗)))((2)(a)∗)/

no, we do not need and extra symbol for unary negation: it can be represented as -(0)(b) and (0)(b)- in prefix and postfix notation respectively (corresponding to 0-b in infix notation)

7>

consider the following program:

```
# include <stdio.h>

int main(){
    int x =0;
    printf("% x % x\ n",& x,& (& x));
    return 0;
}
```

compilation output:

test.c: In function ‘main’:

test.c:5:22: error: lvalue required as unary ‘& ’ operand

printf("% x % x\ n",& x,& (& x));

The error reported is that the & operator takes in only an l-value as an argument and & x always returns a constant hexadecimal literal, hence this expression can never be valid in c irrespective of what x represents in the above program.

8>

No, this is not a coincidence as when designing a language based on the reference model, problems such as memory leaks and dangling references are faced if the task of freeing the allocated memory is left to the programmer. To deal with this, an automatic garbage collector is a must for saving on the programmer's time and effort resulting in a more convenient experience while coding, allowing the programmer to focus on the core project at hand rather than secondary details which should be an important objective for any language.

24>

The objective of this piece of code is to find the first row with all entries as zeros; if it exists, the index of the row is returned otherwise the variable first_zero_row holds -1 as its value.

No, this use-case is not convincing as one can do without using the goto as shown in the code below and with more readable code.

```
int first_zero_row = -1
int i,j
for(i=0;i<n;i++){
        for (j=0;j<n;j++){
                if(A[i][j]) break;
        }
        if(A[i][j]) continue;
        first_zero_row=i;
        break;
}
```

the code above breaks off the inner loop when first encountering a 1 in the row and proceeds on checking the next row if it exited the inner loop in this fashion. If it did not exit the loop this way(all zeros in the row), it sets the first_zero_row variable to the correct index and exits the outer loop displaying the same behaviour as before without the need of a goto also making the code more readable.

Chapter 7 : CHECK YOUR UNDERSTANDING

2>

strongly typed: a language is strongly typed if it doesn't allow, in some way achievable by the language implementation, the application of any operation to any entity that is not intended to be operated in that manner.

statically typed: a language is statically typed if it is statically typed if it is strongly typed and type checking can be performed at compile time. Type checking is the process of ensuring that the program abides by the language's type compatibility rules. A violation is said to be a type clash.

Because C is a more von-neumannish language, it is closer to the hardware and the programmer has more flexibility with how to deal with the bit patterns represented by entities: for instance, in pointer arithmetic, ptr+1 , doesn't increase the hexadecimal value of ptr by 1 but by the number of bytes the value to which ptr points to can be stored in. As it is architecture dependent, enforcing universal type checking rules gets difficult.

7>

C and Icon:

C uses integers in the place of Booleans where a 0 represents false and anything else is true.

Icon replaced Booleans with a generic idea of failures and successes using a generator mechanism.

10>

aggregates are composite literals. They come in handy when we need to initialize, say a user-defined type such as a struct which is a combinational of multiple fundamental types. They can be positional or key-worded. They save a lot of time when initializing composite data types.

11>

Two types are said to be equivalent if they are "equal". The "equal" can be interpreted in two ways: Structural equivalence and Lexical equivalence. Structural equivalence is based on the content of the type definition. Name/Lexical equivalence is based on the lexical occurrence of these type definitions (this results in each definition creating a new type). Structural equivalence allows the language designer to define the extent to which attributes of a type definition should be checked for them to be "equivalent".

Type compatibility is a loser constraint than type equivalence. Again, its definition is language dependent. Two types are compatible if they can be operated with that operator and results and side-effects make sense: for instance, an integer and a double

are compatible with each other when being added together.

15>

Type conversion (casting) changes the value of one type to another

Type coercion performs a conversion automatically in some contexts

Nonconverting type casts are used to interpret the bit pattern of a value according to a different type's guidelines.

19>

Type inference is performed when the type of temporaries (expressions, for instance) or a variable need to be evaluated depending on the context.

Some instances when types are inferred rather than being explicitly mentioned:

the sum of two integers will be an integer

the result of a comparison will be a Boolean

the result of an assignment(where they are expressed by expressions) will be the type of the l-value in the assignment

23>

An assigment can be represented by a single operation between the two records where, it is easier to do so depending on the memory layout used by the language to represent the record.

However, comparison(equality for instance) requires all fields of the record to compared one by one and is inherently more tedious. Hence, languages like C avoid the situation by outlawing full-record comparisons and one has to write relevant routine themselves for the desired purpose.

26>

Uses of variant records(Unions):

allowing the same set of bytes to interpreted in two different ways

represent alternative sets of fields within a record which allows for an efficient way to save on memory which will matter in the case of large databases, for instance.

33>

Contiguous allocation gives the programmer the freedom to exploit cache benefits by providing two ways to store the arrays: row-major and column-major : the way matters when the order in which the matrices will be accessed differs predictably.

Row-pointer layout is used by some languages as an alternative to Contiguous alloca-

tion. The subarrays(rows) can lie in segmented portions of the memory and we have a corresponding array pointing to the head of different rows. This ofcourse results in a bit more space required by Row-pointer method. However, it allows the programmer to use Jagged-arrays (rows of different lengths) (also called ragged arrays) and this also allows for constructing a new array from pre-existing arrays without copying the original ones to a different memory location.

34>

In Row-major order, consecutive locations in the memory hold elements that differ by one in the final subscript (except the last element of the row) i.e. rows are concatenated one after another in the memory when storing them.

In Column-major order, consecutive locations in the memory hold elements that differ by one in the initial subscript (except the last element of the column) i.e. columns are concatenated one after another in the memory when storing them.

The programmer must know the difference between the two storage types and which one the compiler uses as programs that use nested loops to access all the elements of large, multidimensionally arrays vary in performance depending on the compiler's choice. The incorrect combination of code and compiler inference could result in a lot of cache misses during run time resulting in the slower execution of the code.

42>

Pointers and arrays go hand-in-hand in C. Generally, the array name in C is automatically converted to a pointer to the first element of the array. This allows for the programmer to use efficient pointer arithmetic. However, this requires the array to be stored contiguously and failure to do so occasionally results in a false dereference (segmentation fault). In case of the array being stored on the stack and assuming the stack is filled even after the array, the programmer would not even know in the case of an out-of-bound access if the new element being pointed to is the same type.The risk of this is higher in case we are using row-pointer method for storing arrays as C-compilers are not forced to check if an array is not segmented and the programmer chooses the contiguous way to reference to elements such as $*$ (a + n_rows$*$ i + j) rather than $*$ ($*$ (a+i)+j). With that out of the way, pointer arithmetic give power to the programmer to perform some quick elementary operations of the array and the results are scaled by the size of data type help by the array which is convenient.

45>

Garbage refers the sections in the /relating to memory when executing a process regarding to the memory leaks and dangling references that need to be handled in order to free the resources consumed by them and avoid any unpredictable behaviour in case the program tries to access them in incorrect ways.

Garbage is created when the user forgets to free any heap-allocated memory(memory leak) or when a pointer still point to the un-bound address after freeing the corresponding memory

Reference counts: a count is maintained on the number of references that point to the object of concern and whenever the count reaches zero, the object's resources are reclaimed by the process(freed). However, a memory leak occurs when one can't access an object from a named reference i.e. a pointer residing on the stack to be specific. Hence, in the case of dynamically allocated circular lists, the list's nodes won't be reclaimed even if the head of the list (located in the stack) starts pointing to another object or is terminated at the end of a function's scope. With that in mind, it is relatively easy to implement this mechanism.

Tracing collection: this has a better definition of a useful object being one that is referred to by a named variable (something outside the heap). Trace collectors then work by recursively exploring the heap, starting from terminal pointers and deallocating the object if one doesn't reach a non-heap location for a reference. This is more taxing (computationally) and introduces non-uniformity in the run-time performance depending on when the garbage collector decides to kick in.

46>

Mark-and-Sweep

this involves traversing the heap twice: once to mark the useless blocks(found by a recursive traversal back to references).

the second traversal deletes the useless blocks (frees the corresponding resources for use by other processes).

Stop-and-Copy

this algorithm deals with the problem of memory fragmentation. The heap is divided into two virtual halves and memory is allocated in a particular half. When this half is nearly full, the collector kicks-in and each reachable block in this half is copied to the other in a compact manner, simultaneously updating the references and marking the complete first half as free for use.

This means that only half of the heap can be used at once, however. This also avoids the need to perform the first and last steps of the mark and sweep algorithm (finding which ones are useless and deleting them) and instead just copies the useful blocks to the second half. The role of the halves is then switched the next time the collector is called.

Generational Garbage Collection

For further reducing the cost of collection, the heap is segmented into generations which correspond to the objects with different life-expectancies. Each newly allocated object starts-off in a nursery segment and is promoted to the next generation once it passes the collector's examinations without being freed. It is shares similarities to both mark-and-sweep and stop-and-copy in the sense that it deletes and copies the segments in different situations. Whenever the process is low on storage, it first only frees the useless blocks in the nursery segment and proceeds to the next generations only if necessary, thus saving on time.

6>

consider A = 1 and B = -2 ;

showing the behaviour of rem and mod for these values:

rem:

1 = (1/-2)∗ (-2) + (1 rem -2)

(1 rem -2) = 1 - 0 = 1

mod:

1 = (-2)∗ (-1)+(1 mod -2)

(1 mod -2) = 1 - 2 = -1

Similarly, any generic case where A and B are of opposite signs will yield differing results

Talking about the use cases of the two types.

Now say when checking if a number is odd, using the program:

```
bool is_odd(int n){
        return (n mod 2==1)
}
```

will do the job for any n but using rem in this case i.e. :

```
bool is_odd(int n){
        return (n rem 2==1)
}
```

is not sufficient as negative numbers will not pass the test and will return False as the predicate value (-1 != 1)

However, in Mathematics (number theory), the remainder/modulo is considered to be positive irrespective of the signs of A and B and these discrepancies arise only due to different implementations. So, it is better to have only one modulo operator that returns only the absolute value of the corresponding result and the programmer should explicitly handle the sign according to the needs of the situation rather than remembering which one follows what kind of division (truncated and floor).

Both C and Pascal use the first implementation i.e. the answer is dependent on the sign of A only. I think this is the right choice because programs usually have the B hard coded (dividing the expected output in equivalence classes of our choice) and A varies at run time (as in the is_odd check above)

15>

The size of the struct will be 8 bytes. Assuming that A is stored in Row-major order(not possible to answer if it stored in row pointer order)

the address of A[3][7] will be 1000 + 8∗ (3∗ 10 + 7) = 1296

17>

contents of registers initially:

r1: A

r2: i

r3: j

desired final state : r1 contains the value of A[i][j]

cases: storage using row-major format and row-pointer format

Row-major format: A[i][j] = ∗ (A + 4∗ (9∗ i +j))

simple as integers are stored contiguously

Row-pointer format: A[i][j] = ∗ (∗ (A+8∗ i) + 4∗ j)

here i corresponds to the index of the row i.e. it will be a hexadecimal pointer taking up a storage of 8 bytes. Hence, ∗ (A+8∗ i) will contain the pointer to the ith row and increment by 4∗ j to reach the jth column.

Definition of the pseudo language:

(defining only what is needed)

multiply:

mul r1,r2,r3:-

side effect : r1 = r2∗ r3

add:

add r1,r2,r3:-

side effect : r1 = r2+r3

load:

ld r1,r2 :-

side effect : store contents at memory address r2 into r1

Row-major format:

mul r2,r2,9

add r2,r2,r3

mul r2,r2,4

add r2,r2,r1

ld r1,r2

Row-pointer format:

mul r2,r2,8

add r4,r1,r2

ld r2,r4

mul r3,r3,4

add r2,r2,r3

ld r1,r2

The code for row-major is likely to be faster as we have a single load instruction in that case instead of two loads when using a row-pointer format, the rest of the instructions being the same in number.

20>

double ∗ a[n]:

a is an array containing n pointers to doubles

double (∗ b)[n]:

b stores the address of the first element of an array containing n doubles

double (∗ c[n])():

here we are declaring a function which is stored at the address ∗ (c+n) and the function returns a double.

double (∗ d())[n]:

here an array of n doubles is being declared and the pointer to the first location is stored as the address location returned by the function d().

49>

Java has 4 types of references which are used to distinguish between the treatment received by them from the Garbage collector: Strong, Weak, Soft and Phantom.

If an object has only weak references (they have to be explicitly declared), it is marked for garbage collection. The default one is strong which works like a normal reference. Weak references can allow us to use the mark and sweep algorithm or any corresponding algorithm from the same family to make all references from a pointer i n the heap data to be a weak one: that way we can deal with issues such as circular queues where only the head will be a strong reference and the links between nodes will be weak references.