Fall 2019 Programming Languages Qualifying Exam

CODE_____

This is a closed book test.

Correct, clear and precise answers receive full marks

Please start a new page for each question.

There 7 questions – 90pts total

1) (10pts) Define **named constant.** Describe how named constants are implemented with both static and dynamic binding of values. How might a runtime system ensure that the semantics of a **named constant** be enforced.

A named constant is a avariable that is bound to a value once and adds to the readability and maintenance of the code. The value of the name constant is established either at load time which would be static, or can be bound at runtime once (dynamically bound). For compiled languages that are statically bound, the compiler can be enhanced to recognize multiple assignments to that variable. When a named constant is allowed to have dynamic setting, care must be made to eliminate the chance for updating twice like in a select statement or itterative statement. One technique could be to add a secondary variable that states that the variable has been set and then to check for the flag on assignment issuing a runtime error. Additionally, where languages allow aliasing, problems may occur in implementing the named constant correctly. (chapter 5.8 + synthesis)

2) (15pts) Consider the Chomsky Hierarchy.

a) What are the Chomsky grammar types?

b) What does BNF stand for?

c) Identify the Chomsky grammar type (language class) in which BNF would be mapped to.

d) Provide a specific syntactic **programming language construct** that naturally fits the identified grammar type.

a) Chomsky Hierarchy: Regular, Context Free, Context Sensitive, Unrestricted
b) BNF : Backus Nauer Form
c) BNF is included in the Context Free type
d) Matching parenthesis, Matching curly braces, matching square braces, etc. Anything of the for of A^NB^N

3) (15 pts) Write a recursive decent parser functions for <factor> and <expr> for the following BNF fragment. You may assume **next_token()** and **consume_token()** are already implemented. **next_token()** only reads the next token, while **consume_token()** consumes the token. Both methods return a token.

<factor> → <exp> ** < factor>
| <exp>

<exp> → **‘(‘** <expr> **‘)’**
| **id**

Where { **id, **, (,)** } are tokens

```
boolean factor() {
    if ( expr() )
      {
        if (next_token == ‘**’)
          {  consume_token();
           return factor();
          }
        else
          return true;
      }
   else return false;
}

boolean expr() {
  if (next_token == id )
     return true;
  if (next_token() == ‘(‘
     {  match_token();
        if ( factor() )
         {  if (next_token() == ‘)’
               return true;
            else return false;
         }
        else return false;
     }
}
```

4) (10 pts) Using a functional language, like LISP, write a program which sums up the total number of numbers in a list. You may assume the list only includes numbers. For example:

(sum ‘()) = 0
(sum ‘(1 2 3 4) =10
(sum ((1 2) (3 (4 5)))) = 15

```
(define (sum L)
  (cond ((null? L) 0)
        ((not (pair? L)) L)
        (else (+ (sum (car L)) (sum (cdr L))))))
```

5) (15pts) Some Object Oriented (OO) languages use only Objects, while other OO languages use mixed objects and non-object entities. Describe the differences in the approaches. Give examples and explain how runtime systems handle these differences.

The Java language for example has primitives as well as objects. The use of primitives makes Java code (potentially) run faster as it allows the compiler to directly map these elements to the processor and select the appropriate op codes. By allowing primitives, it means that Java must distinguish between the two when dealing with objects that only deal with reference variables. For example, the use of generic classes (like a Stack) cannot push primitive types. This means that Java implements mechanisms to have both a primitive type "int" as well as an object type "Integer". Which then makes the language less readable.

Alternatively, languages that use only Objects (like python) do not suffer from dealing with elements that may not be a reference variable. The trade off for these languages it that all types that map to a processor (like adding two integers) requires additional dereferencing to determine the base values to send to the ALU on the CPU.

6) (10 pts) Provide the weakest Precondition for the following statement

```
if ( x > 4)
    x = x * 2 – 5
else
    x = x + 4
```

{ x < -10 .or. x > 6 }

{ x < -14 .or. 4 >= x > 2 .or. x > 5}

7) (15pts) Dynamic Heap Memory Management

a) Describe and give examples of the difference between a dangling pointer (or reference) variable and a lost heap-dynamic variable as it relates to the heap.

Dangling pointers and lost head variables deal with allocation of memory in the heap.
Dangling pointers

```
int *p = malloc(100);
int *q = p;
free (p);
```

At this point, q may point to freed memory in the heap. The heap management system can keep a secondary incident memory cell on many references are there, or just reallocate the space making the space "q" points to different (and causing unreliability to the code).

A lost variable comes from abandoning the allocated space. For examples

```
int *p = malloc(100);
p = malloc (200);
```

The original 100 bytes is no longer being referenced. The heap management system may need to reclaim this. Techniques include keeping a link count or doing a "mark-sweep" process marking all reaching memory cells.

b) Consider a runtime system which only allocates two different sizes from the heap. S_1 and S_2 where S_1 is twice the size of S_2. Describe a heap memory management algorithm which reduces (eliminates) the need to perform total memory compaction.

There can be several straight forward solutions to this problem. A basic process is that to cut the heap into S_1 and put them onto a "Large" free list. The "Small" free list will either be empty or one element. When a S_2 is request, a free S_1 element is cut in half and the one half is allocated to the program and the other half is kept in reserve. A second request for S_2 would then get the second half.

Reclamation would still require a mark-sweep or extra link count variable. Whenever there is a need for an S_1 segment and there is only a lot of S_2 elements, then two S_2 segments need to be collocated by copying the data from its "buddy" to the other open segment making an S_1 larger element. This requires that the memory management system knows which variables are pointing to each of the S_2 or by maintaining a front end virtual mapping between program variable to actual heap segment.