NJIT

THE EDGE IN KNOWLEDGE

CS 280
Mid-term Exam
Review Examples and Exercises

Spring 2023

Mid-term Exam

- Chapters included:
  - Ch.1
  - Ch. 3: 3.1-3.4
  - Ch. 4: 4.1-4.4
  - Ch. 5: 5.1-5.6
  - Ch. 6: 6.1-6.5 (excluding 6.5.7), 6.7, 6.11 (excluding 6.11.7), and 6.13
  - Ch. 7: Ch. 7.1-7.2 (excluding 7.2.1.4-7.2.1.5), 7.3-7.8

- Type of Questions Expected
  - Multiple-choice (22, 2 points each)
  - True/False (16, 1 point each)
  - Total points: 60

Ch. 1 Examples

- Which produces faster program execution, a compiler or a pure interpreter.
  - pure interpreter
  - Either one of them
  - Compiler
  - Hybrid implementation system

- What is the programming language category whose structure is dictated by the von Neumann computer architecture?
  - Imperative.
  - Logic
  - Functional
  - Object oriented

Ch. 3 Examples

- Given the following grammar in BNF:

  <assign> → <id> = <expr>

  <id> → A | B | C

  <expr> → <id> + <expr>

  | < id> * <expr>

  | (<expr)

  | < id>

  Show a parse tree and a leftmost derivation for the following statement: B = C * (A * C + B)

Ch. 3 Examples

B = C * (A * C + B)

Derivation:

```
<assign> => <id> = <expr>
         => B = <expr>
         => B = <id> * <expr>
         => B = C * <expr>
         => B = C * (< expr> )
         => B = C * (< id> * <expr> )
         => B = C * ( A * <expr> )
         => B = C * ( A * <id> + <expr> )
         => B = C * ( A * C + <expr> )
         => B = C * (A * C + <id> )
         => B = C * (A * C + B )
```

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
       | < id> * <expr>
       | (<expr)
       | < id>
```

Ch. 3 Examples

- What is the correct EBNF for the following BNF rule?

```
term ::= term * factor
             | term / factor
             | term % factor
             | factor
```

a. term ::= term (* | / | %) factor
b. term ::= term {(* | / | %) factor}
c. term ::= factor {(* | / | %) factor}
d. term ::= factor (* | / | %) factor

Ch. 3 Examples

- Given the following BNF grammar for a language with two infix operators represented by # and $.

  Foo ::= Bar $ Foo | Bar

  Bar ::= Bar # Baz | Baz

  Baz ::= x | y | (Foo)

a) Which operator has higher precedence: (i) $; (ii) #; (iii) neither; (iv) both
b) What is the associativity of the $ operator: (i) left; (ii) right; (iii) neither
c) What is the associativity of the # operator: (i) left; (ii) right; (iii) neither
d) Assuming that the start symbol is Foo, is this grammar: (i) ambiguous or (ii) unambiguous?
e) Give a parse tree for the following string: x $ x # y # (y $ x)

Ch. 3 Examples

- Given the sentence: x $ x # y # (y $ x)
- Derivation steps based on leftmost derivation

Foo => Bar $ Foo => Baz $ Foo => x $ Foo => x $ Foo => x $ Bar

⇒ x $ Bar # Baz => x $ Bar # Baz # Baz => x $ Baz # Baz # Baz = x $ x # Baz # Baz

⇒ x $ x # y # Baz => x $ x # y # (Foo) => x $ x # y # (Bar $ Foo)

⇒ x $ x # y # (Baz $ Foo) => x $ x # y # (y $ Foo) => x $ x # y # (y $ Foo)

⇒ x $ x # y # (y $ Bar) => x $ x # y # (y $ Baz) => **x $ x # y # (y $ x)**

- Repeat the derivation for the same sentence based on leftmost derivation by selecting an alternative RHS to show whether the grammar is ambiguous.

Ch. 3 Examples

- Given the following grammar with nonterminals *S*, *A*, and *B*?

  S → A a B b

  A → A b | b

  B → a B | a

  Which of the following sentences is a valid one in the language generated by this grammar.

  a. **baab**
  b. **bbbab**
  c. **bbaaaa**
  d. **bbabb**

Ch. 4: Lexical Analysis

- A lexical analyzer is a pattern matcher for character strings
- A lexical analyzer is a “front-end” for the parser
- Identifies substrings of the source program that belong together - *lexemes*
  - Lexemes match a character pattern, which is associated with a lexical category called a *token*
    - `sum` is a lexeme; its token may be `IDENT`
- The lexical analyzer is usually a function that is called by the parser when it needs the next token.

Matching Strings to Regular Expressions

RegExpr	**Meaning**
x	a character x
\x	an escaped character, e.g., \n
M | N	M or N
M N	M followed by N
M*	zero or more occurrences of M
M+	One or more occurrences of M
M?	Zero or one occurrence of M
[*characters*]	choose from the characters in []
[aeiou]	the set of vowels
[0-9]	the set of digits
. (that’s a dot)	Any single character

You can parenthesize items for clarity

Ch. 4 Examples

- Regular Expressions: Creating expressions given a description.
  - A sequence of digits of any length that also contains the sequence of the five letters “hello” in that order, anywhere in the sequence.
    - **[0-9]*hello[0-9]***
  - A string that begins with a G and ends with an !, with one or more letters in between.
    - **G[a-zA-Z]+!**

Ch. 4 Examples

- Regular Expressions: Matching strings to regular expressions
  - Given a regular expression and a string, indicate the character in the string where the mismatch happens:
  - [A-Z]+y?: "ENyce"
    - Solution: the lowercase letter 'c'.
  - \-?[1-9][0]+: -5
    - The string ends before the pattern
  - ([A-Z][A-Z])+: GOOOOOGLE?
    - The '?' in the string

Ch. 4 Examples

- Which regular expression that matches a sequence of a string of zero or more even digits of any length followed by one or more letters?
  a. [0 2 4 6 8]*[a-zA-Z]+
  b. [0 2 4 6 8]+ [a-zA-Z]+
  c. [0 2 4 6 8][a-zA-Z]
  d. [0 2 4 6 8]?[a-zA-Z]*

Ch. 4 Examples

- Why does the following grammar rules cause a catastrophic problem for recursive-descent parser?
    1. $E \rightarrow E + T \mid T$
    2. $T \rightarrow T * F \mid F$
    3. $F \rightarrow (E) \mid id$

a. Rules (i) and (ii) have direct left recursion.
b. The grammar is ambiguous.
c. The grammar rules need to enforce operator precedence.
d. The grammar rules do not pass the disjointness test.

Ch. 4 Examples

- Prob. 1 (p.193): Perform the pairwise disjointness test for the following grammar rules
  a. `A -> aB | b | cBB`
  b. `B -> aB | bA | aBb`
  c. `C -> aaA | b | caB`
- Solutions
  a. `FIRST(aB) = {a}, FIRST(b) = {b}, FIRST(cBB) = {c}, Passes the test`
  b. `FIRST(aB) = {a}, FIRST(bA) = {b}, FIRST(aBb) = {a}, Fails the test`
  c. `FIRST(aaA) = {a}, FIRST(b) = {b}, FIRST(caB) = {c}, Passes the test`

Ch. 5: Categories of Variables by Lifetimes

- Static--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution.
- Stack-dynamic--Storage bindings are created for variables when their declaration statements are *elaborated*.
- *Explicit heap-dynamic* -- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
- *Implicit heap-dynamic*--Allocation and deallocation caused by assignment statements.

Ch 5: Categories of Variables by Lifetimes

- The storage binding of all declared local variables in C++ functions are of the category of ____________________.
  a. Stack-dynamic variables.
  b. Explicit-Heap dynamic variables.
  c. Implicit Heap-Dynamic variables.
  d. Static variables.

- All created class objects in Java must be allocated memory storage ___________________.
  a. Implicitly dynamic from the heap.
  b. Dynamically from the run-time stack.
  c. Explicitly from the heap using the `new` operator.
  d. Statically from the data segment by the compiler.

Ch. 5: Categories of Variables by Lifetimes

```
int outside;    // external variable
                // exists for lifetime of the program
void myFunction() {
  static int counter=0;
  // exists for lifetime of the function call
  int val
  SomeClass x;
  SomeClass *p;

  // the memory returned by new comes from the heap
  // exists until cleaned up/freed (by user or by runtime
  // environment)
  p = new SomeClass();
}
```

Ch. 5: L-values and R-values

- Given the following declaration,

```
int *ptr;
int x = 5;
```

What is the *r-value* of *ptr* in the following statement?

```
ptr = & x;
```

a. L-value of x variable.
b. R-value of x variable.
c. Pointer variables do not have an r-value.
d. R-value for ptr has not been defined.

Ch. 5 Examples

- Given the following statements:

  ```
  int *P, z = 7;
  . . .
  P = & z;
  ```

  What is the l-value of `P`? What is the r-value of `P`? What is the r-value of `z`? and what is the l-value of `z`?

  - L-value of P is the location of P in memory.
  - R-value of P is the address of z, or the l-value of z.
  - R-value of z is its contents, the value of 7.
  - L-value of z is its address location in memory.

Ch. 5: Scope

- Variables can be hidden from a unit by having a "closer" variable with the same name
  - JavaScript example of nested functions:

```
function big() {
  function sub1() {
    var x= 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

Ch. 5: Scope

- Example in C/C++: Nested blocks

```
void sub() {
 int count;
 while (...) {
int count;
   count++;
   ...
 }
 …
}
```

Ch. 5: Scope

- Consider the following C++ function definition,

```
void function(void){
 int a, b, c;//definition 1
 . . .

 while (. . .) {
   int b, d;//definition 2
   . . .
   . . .
   if (. . .) {
        int e, a;//definition 3
        . . . //Point 1
   }
        }
}
```

a. Variables a and e from definition 3, variables b and d from definition 2, and variables c from definition 1.
b. Variables a and e from definition 3, and variables b and d from definition 2.
c. Variables a and e from definition 3, and variables b and c from definition 1.
d. Variables a and e from definition 3 only.

Determine the visible variables at Point 1, inside the if statement of the function, using the labelled definition statements by comments in the function,

Ch. 5: Scope

```
function big() {
    function sub1(){
      var x = 7;
      . . .
      sub2();
    }
    function sub2() {
      var y = x;
           . . .
    }
   var x = 3;
    ...
    sub1();
 }
```

```
big calls sub1
sub1 calls sub2
sub2 uses x
```

- Static scoping
  - Reference to `x` in `sub2` is to `big`'s `x`
- Dynamic scoping
  - Reference to `x` in `sub2` is to `sub1`'s `x`

Ch. 6: Definitions

1. What is a descriptor?
2. Define row major order and column major order.
3. What is an access function for an array?
4. What are the two common problems of pointers?
5. What is a C++ reference type, and what is its common use?
6. What is a compatible type?
7. Define type error.

Ch. 6: Primitive Data Types

- Which of the following primitive data types is not a reflection of the hardware, i.e., it is not supported by hardware implementation.
  a. Integer
  b. Floating-point
  c. Character
  d. Decimal

Ch. 6: Enumerated Types

- Given the following C++ definition and declarations:

```
 enum Season { Summer, Fall, Winter, Spring};
Season thisSeason;
int res;
```

Which one of the following statements would generate a syntax error?

```
a. thisSeason = Winter;
b. res = Spring;
c. thisSeason = (Season) 2;
d. thisSeason =  2;
```

Ch. 6: Arrays

- Given the following two-dimensional array declaration in C++

  int table [5][8];

  Assume row major order allocation in memory, where an integer variable is allocated 4 bytes and the address of table array is 100. What is the address of the element at table[3][4]?

  Solution:

  Addr table[3][4] = 100 +(3*8 + 4) * 4 = 212

  a. 212
  b. 244
  c. 208
  d. 216

Ch. 6: Pointers

- Consider the following declarations, indicate by True/False the following:
  - *(ptr + 1) is equivalent to list[1]
  - *(ptr + index) is equivalent to list[index]
  - ptr[index] is equivalent to list[index]

```
int list [10];
int * ptr;

ptr = list;
```

Ch. 6: Pointers

- What is a dangling pointer?
  - A dangling pointer is a pointer that contains the address of a heap-dynamic variable that has been deallocated.
  - Example:
    - Now arrPtr1 and arrPtr2 are dangling pointers
- Garbage variable is a lost heap-dynamic variable that is no longer accessible to the user program. (True/False)

```
int * arrPtr1;
int * arrPtr2 = new int[10];

arrPtr1 = arrPtr2;

Delete [] arrPtr2;
```

```
int *ptr1 = new int;
*ptr1 = 5;
int * ptr2 = new int;
*ptr2 = 7;
ptr1 = ptr2;
cout << *ptr1 << " " <<
       *ptr2 << endl;
//first heap-dynamic
//variable is garbage now.
```

Ch. 6: Reference Types

- Given the following declaration,

```
int value = 0;
```

  - A **C++ reference variable**, valRef, needs to be declared and initialized with the address of the variable value. Which of the following options is the correct declaration statement for the `valRef`.

```
a. int &valRef = value;
b. int &valRef = &value;
c. int *valRef = value;
d. int *valRef = &value;
```

CH. 7: Definitions

1. Define operator precedence and operator associativity.
2. What is a prefix operator?
3. Define functional side effect.
4. What is coercion?
5. Define narrowing and widening conversions.
6. What is an overloaded operator?
7. What is a mixed-mode expression?
8. What is short-circuiting evaluation?

Ch. 7: Examples

- **Expressions**
  - Assume the following rules of associativity and partial precedence of operators in C-based language for expressions:

Precedence	Highest	unary +, unary -, !
		*, /, % (mod)
		+, -
		<, >, <=, >=
		==, !=
		&&
	Lowest	||

Associativity	**Left to Right**	

Ch. 7: Examples

Evaluate the following C++ expressions based on the assumed associativity and precedence rules given.

Assuming the following declarations.

```
int i= 10, j= 4;
double m= 2, n= 4;
```

Use the given table of precedence rules to evaluate the following expression, assuming associativity is left to right.

i = i % j – i * j / (m – i) + m;

a. 9
b. -1
c. 8
d. 12

Ch. 7: Examples

- **Functions side effects**
  - Consider the following C++ program:

    What is the value of x after the assignment statement in main function, assuming:

    a. Operands are evaluated left to right.

    b. Operands are evaluated right to left.

```
int fun(int *i){
 *i += 5;
 return 4;
}
int main(){
  int x = 3;
  x = x + fun(&x);
  cout << x << endl;
  return 0 ;
}
```

  - Results Case 1

    a. 7 (actual run is 12)

    b. 12

  - Results Case 2

    a. -1

    b. 4

```
int fun(int *i){
 *i += 5;
 return 4;
}
int main(){
  int x = 3;
  x = x - fun(&x);
  cout << x << endl;
  return 0 ;
}
```

Ch. 7: Examples

- **Assignment as an expression in C/C++**
  - What is the value of x after executing the following code in C/C++?

```
int x = 3, y = 5, z;
z = x;
if(x = y)
    x += y;
else if(y > x)
    x = y % x;
```

a. 5	**b. 3**	**c. 2**	**d. 10**

Ch. 7: Examples

- Given the following C++ declarations.

```
int val = 2;
float res = 3.5;
```

Indicate which of the following operations is a narrowing type conversion.

```
a. val = res;
b. res = val;
c. (float) val
d. (double) res
```

Ch. 7: Examples

- Prefix increment and decrement operators (e.g., ++x, --x) in C++ has a precedence level that is

  a. Lower than multiplication and division arithmetic operators
  b. Higher than multiplication and division arithmetic operators
  c. Lower than multiplication and division operators and higher than addition and subtraction operators
  d. Lower than addition and subtraction operators.

NJIT

THE EDGE IN KNOWLEDGE