Wentworth Institute of Technology

COMP 3350 Programming Languages - Final Exam

**Answer any 10 questions. Best 10 will be selected, if you answer more than 10**

1. Match the definitions [10]
2. Type inference 1. It cannot support recursive subprograms.
3. Formal parameters 2. Pattern matcher that isolates the small

parts of a program

1. Static variables 3. Type that is either legal for an operator or

is allowed to be implicitly converted to a

legal type.

1. Compatible type 4. Place holders for params in a function  
    definition.
2. Lexical Analyzer 5. Context is the type of the value assigned

To the variable in a declaration statement

1. Fill in the blanks: [10]
   1. Storage binding for a static variable happens at \_\_\_\_\_\_\_\_\_\_\_ and type binding happens at \_\_\_\_\_\_\_\_\_\_\_\_\_ (Load time, compile time, run-time)

* 1. Storage binding for a heap-dynamic variable happens at \_\_\_\_\_\_\_\_ and type binding happens at \_\_\_\_\_\_\_\_\_\_\_ (run-time, load-time, compile time)
  2. An array organization which places the whole first column in memory, followed by the second column, and so on is called \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. (column-major order, row-major order).
  3. Explicit heap-dynamic variables are often used to construct dynamic structures using a \_\_\_\_\_\_\_\_\_\_\_\_\_ (pointer, static array, reference).
  4. A \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ array is one in which subscript ranges are statically bound, but the allocation is done at declaration time during execution (static, explicit heap-dynamic, fixed stack-dynamic, fixed heap-dynamic)

1. True or False? [10]
   1. A data type, such as int in C, is bound to a range of possible values at compile time. (**T** / F).
   2. A variable’s value is sometimes called its r-value (**T** / F)
   3. An enumeration type (in C, C#, C++) has all its possible values implicitly assigned. They cannot be changed. (T / **F**)
   4. The asterisk symbol (\*) is bound to the multiplication operator at language design time. (**T**/F)
   5. In a language with dynamic scope, the bindings between names and objects can be determined at compile time by examining the text of the program. (**T**/F)
2. Consider the following program written in C syntax: [10]

void swap (int a, int b) {

int temp;

temp = a;

a = b;

b = temp;

}

void main() {

int value = 2, list[5] = {1,3,5,7,9};

swap(value, list[0]);

swap(list[0], list[1]);

swap(value, list[value]);

}

For each of the following parameter passing methods, what are all of the values of the variables, value and list, after each of the three calls to swap?

1. Passed by value

None of the arguments are changed.

The variables keep the values they had been initialized with.

1. Passed by reference.

* Value == 1 and list[0] == 2. -> First call
* List[0] == 3 and list[1] == 2. -> Second call
* Value == 2 and list[1] == 1. -> Third call

1. Here is an unambiguous grammar for a simple language with an if statement. The grammar treats s1, s2 as statements and e1, e2 as expressions. [10]

<if-stmt> => if <expr> then <full-stmt> else <stmt>

| if <expr> then <stmt>

<full-stmt> => <full-if> | s1 | s2

<full-if> => if <expr> then <full-stmt> else <full-stmt>

<expr> => e1 | e2

List the start symbol, non-terminals, and terminals.

Start->Symbol: <if-stmt>

Non-terminals: <Full-stmt>, <Full-if>, <expr>

Terminals: <if, then, else, s1, s2, e1, e2>

1. Describe, in English, the language defined by the following grammar in BNF [10]

<S> -> <A> <B> <C> a’s b’s c’s

<A> -> a <A> | a One or more a’s

<B> -> b <B> | b One or more b’s

<C> -> c <C> | c One or more c’s

1. Write a grammar for the language consisting of strings that have *n* copies of the letter *a* followed by the same number of copies of the letter *b*, where *n > 0*. For example, the strings *ab, aabb, aaaabbbb* etc. are in the language but *a, abb, ba*, and *aaabb* are not. [10]

S -> aMb

M-> aMb | epsilon

1. Given a grammar for a simple assignment statement:

<assign> -> <id> = <expr>

<id> -> a | b | c | d

<expr> -> <expr> + <term>

| <term>

<term> -> <term> \* <factor>

| <factor>

<factor> -> (<expr>)

| <id>

Construct a parse tree and an abstract syntax tree for the statement [10]

a = b\*c + d;

<Expr> -> <Expr> \* <Expr> + <Expr>

<Expr> -> <Expr \* Expr> + <Expr>

<Expr> -> <b\*c> + <d>

<Expr> -> <b\*C+d>

<=> -> <\*> <+>

<=> -> <\*> | int

<=> -> <+>| int

<=> -> <b\*C> <+> <d> | int

1. Is there a difference between the scope and lifetime of a variable? Here are two similar C code fragments. In the first code fragment the variable count is declared outside a function. In the second code fragment the variable is defined static inside the function.

**Explain** clearly if the scope and lifetime of the variable count in these two cases is the same or different. [10]

1. int count = 0;

int nextcount() {

count = count + 1;

return count;

}

In this case, the ***count*** variable is the same. As we can see, in the function nextcount(), we’re not passing in a new ***count*** variable. But are referring to the global ***count*** variable in the class.

1. int nextcount() {

static int count = 0;

count = count + 1;

return count;

}

In this case, the ***count*** is not the same as the one above. We now see that ***count*** only exists when nextcount() is called.

To sum it up, the variables aren’t the same.

1. Here is a skeletal C program. The sequence of function calls in this program is given. Also shown is the stack with the 3 Activation records (stack frames).

The activation records for main() and f1() at point 1 are shown. Complete the entries for f2 and f3 at points 2 and 3. [10]

main() calls f1()

f1() calls f2()

f2 () calls f3()

void f1(float r) {

int s t;

…. **1**.

f2(s);

…..

}

void fun2(int x) {

int y;

….. **2**.

f3(y);

…..

} …

void fun3(int q) {

….. **3**. AR for f3() at 3

} …

void main() { q

float p

….. int y

f1(p);

….. f3(y)

} AR for f2() at 2

y

Return to f1()

local variable t

local s

Parameter r

AR for f1() at 1. Dynamic link

Return to main

AR for main local variable p

1. Recall that *Unification* is one of the most fundamental concepts in Prolog. *Terms* are the building blocks of programs and data in Prolog. Two terms which are exactly the same will unify. For example, the atom joe unifies with joe, the number 42 unifies with 42, and the variable X unifies with X – all of these are pairs of identical terms. This also holds for complex terms: happy(joe) unifies with happy(joe). Two different terms will not unify (e.g. jack does not unify with jill).

Since variables have no fixed value, two terms can also unify if they contain variables which may be given values (instantiated) in such a way that the resultant terms would unify by the first part of the definition.

* 1. Which of the following pairs will unify? [5]

1. luke = luke.
2. luke = leia.
3. Luke = luke.
4. jedi(luke) = luke.
5. father(vader, A) = father(B, luke).

Cases i and iv will unify

* 1. The following program defines what it means for a line to be vertical or horizontal: [5]

Vertical (line(point(X, Y), point(X, Z))).

Horizontal (line(point(X, Y), point(Z, Y))).

The point/2 predicate takes two numbers as arguments, which specify the point’s x-coordinate and y-coordinate on the cartesian plane.

They will unify

What will Prolog’s response be to the following queries?

?- horizontal(line(point(16, 4), point(2, Y))).

Yes

Line(point(16,4)

Point(16,4)

Point(2, y)

True? Yes

?- horizontal(line(point(16, 4), P)).

Yes

Line(point(16,4))

Point(16,4)

P

True? No

1. What are the four types of polymorphisms? Here are four simple programs in C/C++/Java. Identify which type of polymorphism each program is an example of.

a)

int square (int x) double square (double x)

{ {

return x \* x; return x \* x;

}

Ad-hoc Polymorphism (Overloading) }

b)

public static int sumEven(List<Integer> li)

{

int sum = 0;

for (Integer i: li)

if (i % 2 == 0)

sum += i;

return sum;

}

Coercion Polymorphism (Casting)

c)

public static <T> int countGreaterThan(T[] anArray,

T elem)

{

int count = 0;

for (T e : anArray)

if (e > elem) // compiler error

++count;

return count;

}

Parametric Polymorphism (Compile-Time)

d)

public class Car {

public void brake() { ….}

}

public class ManualCar extends Car

{

public void clutch() { … }

}

void g(Car z) {

z.brake();

}

void f(Car x, ManualCar y) {

g(x);

g(y);

}

Subtype Polymorphism (Runtime)