Principles of Programming Languages - Midterm (Solutions)

K. Subramani
LDCSEE,
West Virginia University,
Morgantown, WV
{ksmani@csee.wvu.edu}

1 Problems

1. **Language Design Principles:** Using the various languages discussed in class, explain the following concepts:

   (i) Generality.

   (ii) Uniformity.

   (iii) Extensibility.

   (iv) Restrictability.

   Solution:

   (i) Generality - A language achieves generality by avoiding special cases in the availability or use of constructs and by combining closely related constructs into a single more general one.

   For instance, in **C**, structures cannot be compared using the "==" operator; this indicates *lack* of generality.

   (ii) Uniformity - A language achieves uniformity if constructs that look similar have the same meaning and likewise constructs that should behave differently appear dissimilar.

   For instance, in Pascal, both the assignment statement and the value returned by a function use the ":=" construct; the semantics of assignment and function execution are fundamentally different and therefore the above feature of Pascal indicates non-uniformity.

   (iii) Extensibility - A language is said to be extensible, if there exists a general mechanism for the user to add features to a language.

   Any language (C++, Ada) that permits operator overloading is extensible; for instance, matrix addition can be written as integer addition through overloading the "+" operator.

   (iv) Restrictability - A language promotes restrictability, if a programmer can accomplish non-trivial tasks using a minimal knowledge of the language and a minimal knowledge of language constructs.

   For instance, in **C**, it is sufficient to know the syntax and semantics of the **if** and **goto** statements to accomplish looping.

   □

2. **CFG Design:**

   (i) A palindrome is a word that reads the same forward and backward; for instance, "noon" is a palindrome over the English alphabet. Consider the alphabet $\Sigma = \{0, 1\}$. Design a CFG that represents the set of palindromes over Σ. (2 points.)

   (ii) Let $\Sigma = \{0, 1\}$ denote an alphabet. Design a CFG that represents the set of all strings that contain at least one pair of consecutive 0s in them. (2 points.)

Solution:

(i) A CFG for palindromes is given by $G = \langle V, T, P, S \rangle$, where,
   (a) $V = \{S\}$,
   (b) $T = \{0, 1\}$,
   (c) and production rules P are:

$$S \rightarrow 0\ S\ 0 \mid 1\ S\ 1 \mid 0 \mid 1 \mid \epsilon$$

   Proving that the above the CFG indeed derives all and only palindromes is a non-trivial task and beyond the scope of this class.

(ii) Before designing a CFG, we observe that the described language can be captured by the following regular expression: $(\mathbf{0} + \mathbf{1})^* \cdot \mathbf{00} \cdot (\mathbf{0} + \mathbf{1})^*$.
   It is now straightforward to construct the CFG as follows:

$$\begin{aligned} S &\rightarrow S_1 \cdot 0 \cdot 0 \cdot S_1 \\ S_1 &\rightarrow 0\ S_1 \mid 1\ S_1 \mid \epsilon \end{aligned}$$

□

3. **CFG Ambiguity:**

(i) When is a CFG G said to be ambiguous? When is a language said to be inherently ambiguous? (2 points.)

(ii) Let $G = \langle V, T, P, S \rangle$ denote a CFG, with $V = \{S\}$, $T = \{a, b\}$, and production rules P given by:

$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

   Is G ambiguous? Justify your answer with a proof or counterexample. (2 points.)

Solution:

(i) A CFG G is said to be ambiguous, if there exists a string x in $L(G)$, such that there exist two distinct leftmost derivations (or two distinct parse trees) for x from the start symbol. A context-free language L is said to be inherently ambiguous, if *every* CFG G designed for this language is ambiguous.

(ii) G is ambiguous, since the string $w = abab$ has two distinct leftmost derivations:
   (a) $S \Rightarrow aSbS \Rightarrow abSaSbS \Rightarrow abaSbS \Rightarrow ababS \Rightarrow abab$, and
   (b) $S \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSbS \Rightarrow ababS \Rightarrow abab$.

□

4. **Semantics:** Enumerate the differences between:

(i) Lexical and dynamic scoping. (1 point.)

(ii) Static and dynamic type checking. (1 point.)

Is it possible to define the semantics for a language that is dynamically scoped but statically type checked? Justify your answer. (2 points.)

Solution:

(i) In lexical scoping, the attributes of a name can be determined from the text, whereas in dynamic scoping the context of the name occurrence is required to determine its attributes. In other words, in lexical scoping, the scope of a binding is limited to the block in which the associated declaration appears, whereas the attributes of a name in Dynamic Scoping, depend upon the actual execution path chosen by the program. For instance, let x denote a non-local variable in a function $p()$. In lexical scoping, x must have the attributes of the global declaration of x. However, in dynamic scoping, the name x will have the attributes ascribed to it in the calling function. For instance, consider the case in which x is declared in *main*() as a float variable and globally as an integer variable. If $p()$ is called from $main()$, then x will be a float variable in $p()$ under dynamic scoping, but an integer variable under lexical scoping.

(ii) In static type checking, the types of expressions and objects are determined from the text of the program and type checking is performed at compile time, as opposed to run time, whereas in dynamic type checking, the type information is maintained and checked at run time to ensure type compatibility.

It is not possible for a language to be dynamically scoped and statically type checked, since the type attribute of a name depends upon the path of execution in dynamic scoping.

Consider for instance, the expression $x + y$. Under dynamic scoping, the types of x and y could be identical in one execution path, but different in another execution path. Consequently, a static type checker will be unable to decide on the type-appropriateness of an expression, merely by looking at the text, i.e., static type checking is not possible under dynamic scoping. □

5. **Type Theory:** Consider the following block of **C** code:

```
struct A
   {
     int i;
     char j;
    };

 struct B
    {
      int i;
      char j;
     };

struct A x, *p;
struct B y, *q;
```

Which of the following statements lead to static errors? Explain.

(i) `x = y;`

(ii) `x = (struct A) y;`

(iii) `p = q;`

(iv) `p = (struct A*) q;`

Solution:

(i) Static type error since **C** uses name equivalence for structures.

(ii) Static type error since type casting in **C** is restricted to numeric and pointer types.

(iii) Not a static type error, since both p and q are of pointer type; however, a warning will typically be issued to inform the user that there could be problems down the line.

(iv) Not a static type error, since both p and q are pointer variables and the address pointed to by q has been appropriately typecast, before assigning it to p. Indeed, this statement will not even cause a warning.

□