NATIONAL UNIVERSITY OF SINGAPORE
SCHOOL OF COMPUTING

EXAMINATION FOR
Semester 1, 2001/2002
CS2104 - PROGRAMMING LANGUAGE CONCEPTS

November 2001 Time Allowed: 2 Hours

INSTRUCTIONS TO CANDIDATES

1. This examination paper contains **four(4)** long questions and comprises **fifteen(15)** pages.
2. Answer **ALL** questions.
3. Answer **ALL** questions within the space provided in this booklet.
4. This is an **OPEN BOOK** examination. You are allowed to bring in the main text book and the lecture notes. **No other material is allowed.**
5. **Please write your Matriculation Number below.**

MATRICULATION NO.:

(This portion is reserved for the examiner's use only)

Question	Marks	Remark
Question A 14		
Question B 12		
Question C 12		
Question D 12		
Total 50		

A. **Language Syntax and Semantics** (5 + 5 + 4= 14 marks)
As you know, BNF Grammars are a convenient notation for describing program syntax. Consider a programming language PL where any program is of the following form:

begin
declare v_1 : $\langle$type of $v_1\rangle$;
declare v_2 : $\langle$type of $v_2\rangle$;
...
declare v_n : $\langle$type of $v_n\rangle$;
begincode
$\langle$ Sequence of zero or more Assignment Statements $\rangle$
endcode
end

Assume that for PL programs:

- There are zero or more declarations in the declaration section.
- Any variable name is a sequence of lowercase letters.
- the type of each variable is integer or real.
- Each assignment statement assigns an arithmetic expression to a variable, and ends with a ';'
- There is only one operator appearing in expressions: *addition*. This addition operator is denoted by +, and is overloaded.
- Any application of the addition operator (within an arithmetic expression) is between two variables or two real/integer constants or between a real/integer constant and a variable. Thus, if a and b are variables, all of the following are valid expressions: $a + b$, $a + 1$, $1 + 2$.
- The assignment symbol := is used in each assignment statement.

1. Write the complete BNF Grammar which describes the set of all programs which satisfy the above restrictions. Your grammar must not have ambiguity. The start symbol of the grammar should be called *Pgm*.

Answer:

Pgm $\rightarrow$ begin Decl begincode Code endcode end

Decl $\rightarrow$ ϵ | D Decl
D $\rightarrow$ declare Var : Type ;
Type $\rightarrow$ integer | real

Code $\rightarrow$ ϵ | Stmt Code
Stmt $\rightarrow$ Var := Expr ;
Expr $\rightarrow$ Id + Expr | Id
Id $\rightarrow$ Var | Int | Real

Var $\rightarrow$ Letter | Letter Var
Letter$\rightarrow$ a | b | ... | z
Int $\rightarrow$ Minus Posint | Posint
Posint $\rightarrow$ Digit Int | Digit
Digit $\rightarrow$ 0 | 1 | ... | 9
Real $\rightarrow$ Minus Posreal | Posreal
Posreal $\rightarrow$ Int Decimal Int | Decimal Int
Minus $\rightarrow$ -
Decimal $\rightarrow$.

2. Your BNF Grammar allows you to write programs of the form

```
begin
declare a : integer;
begincode a := 1; a := b + 1; endcode
end
```

i.e. a variable appearing in an assignment statement may not be declared. Augment your BNF grammar to an attribute grammar to identify (and disallow) such programs. The nonterminal Pgm should have an attribute *bad* which is true if a program assigns/uses undeclared variables (and false otherwise).

Answer:

Associate an attribute called *varset* with nonterminal Decl (set of variables declared so far). Associate an attribute *allvars* with nonterminal Code (set of all declared variables). The value of varset(Decl) is passed to allvars(Code); allvars(Code) is then passed to the non-terminal Stmt as well. Associate an attribute *bad* with nonterminal Pgm. Similarly, associate attributes *badcode*, *badstmt*, *badexpr*, *badid* with nonterminals Code, Stmt, Expr, Id respectively. The exact computation of the attributes should proceed as:

Pgm $\rightarrow$ begin Decl begincode Code endcode end
$\{\ allvars(Code) := varset(Decl);\ bad(Pgm) := badcode(Code)\ \}$

Decl $\rightarrow \epsilon$ $\{\ varset(Decl) := \phi\ \}$
Decl $\rightarrow$ D Decl $\{\ varset(Decl_1) := varset(D) \cup varset(Decl_2)\ \}$
D $\rightarrow$ declare Var : Type ; $\{\ varset(D) := \{name(Var)\}\ \}$
Type $\rightarrow$ integer | real

Code $\rightarrow \epsilon$ $\{\ badcode(Code) := false\ \}$
Code $\rightarrow$ Stmt Code
$\{\ badcode(Code_1) := badstmt(Stmt) \vee badcode(Code_2)\ ;$
$allvars(Stmt) := allvars(Code_1);$
$allvars(Code_2) := allvars(Code_1)\ \}$
Stmt $\rightarrow$ Var := Expr ;
$\{\ badstmt(Stmt) := badexpr(Expr) \bigvee name(Var) \notin allvars(Stmt);$
$allvars(Expr) := allvars(Stmt)\ \}$
Expr $\rightarrow$ Id + Expr ;
$\{\ badexpr(Expr_1) := badid(Id) \vee badexpr(Expr_2);$
$allvars(Expr_2) := allvars(Expr_1);$
$allvars(Id) := allvars(Expr_1)\ \}$
Expr $\rightarrow$ Id $\{\ badexpr(Expr_1) := badid(Id);\ allvars(Id) := allvars(Expr)\ \}$
Id $\rightarrow$ Var $\{\ badid(Id) := name(Var) \notin allvars(Id)\ \}$
Id $\rightarrow$ Int $\{\ badid(Id) := false\ \}$
Id $\rightarrow$ Real $\{\ badid(Id) := false\ \}$

3. How can you augment the grammar of expressions that you designed in part (1) to disallow type coercion? Discuss. Would this approach be possible if PL programs had no type declarations? How would type coercions be prevented in such a situation ?

   Answer:

   We can change the attribute $varset(Decl)$ compute the set of all declared variables as well as their types. This is then passed to $allvars(Code)$ and from there to $allvars(Id)$. We now associate an attribute *type* with the nonterminal Id. It is computed as follows. The function $get\_type(name(Var), allvars(Id))$ returns the type of Var via table look up.

   $$\begin{array}{l} \text{Id} \rightarrow \text{Var } \{ \ type(Id) := get\_type(name(Var), allvars(Id))\} \\ \text{Id} \rightarrow \text{Int } \{ \ type(Id) := integer \ \} \\ \text{Id} \rightarrow \text{Real } \{ \ type(Id) := real \ \} \end{array}$$

   We now associate a boolean attribute *coerced* with the nonterminal Expr. It is computed as follows:

   $$\begin{array}{l} \text{Expr} \rightarrow \text{Id } \{ \ coerced(Expr) := false; \ type(Expr) := type(Id) \ \} \\ \text{Expr} \rightarrow \text{Id} + \text{Expr } \{ \ coerced(Expr_1) := coerced(Expr_2) \vee type(Id) \neq type(Expr_2) \ \} \end{array}$$

   This approach would not be possible if PL programs had no type declarations, since attribute grammars only capture the static semantics. Then, given a PL program, we cannot compute the attribute *allvars* statically. *One way* to prevent type coercion would be to perform dynamic type checking of operands during expression evaluation.

B. **Logic Programming** (6 + 4 + 2 = 12 marks)
A set is a collection of distinct elements. Certain programming languages like PASCAL support sets as datatypes. This is not done in Prolog. Sets can be manipulated using three operations: union ($\cup$), intersection ($\cap$) and difference ($-$).

- union: the set $S \cup T$ contains only all elements in S and all elements in T.
- intersection: the set $S \cap T$ contains only those elements which appear in both S and T.
- difference: the set $S - T$ contains those elements which appear in S but do not appear in T

1. Define three ternary Prolog predicates `union`, `intersect` and `difference`. Thus, `union(X, Y, Z)` performs the computation `Z := X` $\cup$ `Y`. Similarly `intersect(X,Y,Z)` computes `Z := X` $\cap$ `Y`. and `difference(X,Y,Z)` computes `Z := X - Y`. Use Prolog lists with no repetition to represent sets *e.g.* `[a,b,c]` represents the set { `a`, `b`, `c`}. You may need a predicate `member(X,L)` which checks whether `X` appears in list `L`.

   Answer:

```
union([], Y, Y).
union([X|Xs], Y, Z) :- member(X, Y), !, union(Xs, Y, Z).
union([X|Xs], Y, [X|Z]) :- union(Xs, Y, Z).

intersect([], _, []).
intersect([X|Xs], Y, [X|Z]) :- member(X, Y), !, intersect(Xs, Y, Z).
intersect([_|Xs], Y, Z) :- intersect(Xs, Y, Z).

difference([], _, []).
difference([X|Xs], Y, Z) :- member(X, Y), !, difference(Xs, Y, Z).
difference([X|Xs], Y, [X|Z]) :- difference(Xs, Y, Z).

member(X, [X|_Xs]) :- !.
member(X, [_Y|Ys]) :- member(X, Ys).
```

   Instead of using cuts, we could have used negation also.

2. Prolog has no support for sets as datatypes. Thus, we could ask the query `union([a,a], [b,b], X)` and the underlying implementation will not complain. In fact, Prolog has no support for types at all. So, we could even ask the query `union(mickey, mouse, X)`. Suggest a remedy to this problem (by changing your `union` predicate).

   Answer:

   The solution is to explicitly do type checking at the user level. We can define a set predicate

```
set(L) :- list(L), all_distinct(L, L).

list([]).
list([_|L]) :- list(L).

all_distinct([], _).
all_distinct([X|Xs], L) :- not(member(X, L)), all_distinct(Xs, L).
```

   The implementation of `union` is now changed as follows

```
union_typechk(X, Y, Z) :- set(X), set(Y), union(X, Y, Z).
```

   Similar changes have to be made to `intersect` and `difference`.

3. Reachability in a directed acyclic graph can be computed by the following Prolog predicate:

```
reach(X, Y) :- arc(X, Y).
reach(X, Y) :- arc(X, Z), reach(Z, Y).
```

Assume that `arc(X, Y)` is true when there is an arc from node `X` to node `Y`. If we ask the query `reach(a, b)`, Prolog evaluation will merely tell us whether node `b` is reachable from node `a`. How can we augment `reach` to also return a path between `a` and `b` (if one exists). Thus, you need to define a ternary predicate `reach(X, Y, Path)`. The query `reach(a, b, P)` should fail if `b` is not reachable from `a`; otherwise it should succeed and return a path between `a` and `b` (as a Prolog list).

Answer:

```
reach(X, Y, [X, Y]) :- arc(X, Y).
reach(X, Y, [X|Path]) :- arc(X, Z), reach(Z, Y, Path).
```

C. **Object Oriented Programming** $(3 + 4 + 2 + 3 = 12$ marks)

1. A *stack* is an abstract data type with the operations:

   - S.Push(x) : Push x into the top of the stack S
   - S.Pop() : Removes the element from the top of stack S and returns it
   - S.Top() : Returns the element from the top of S without removing it

   Thus, *stack* is implemented as a class and *Push*, *Pop*, *Top* are implemented as methods. The actual implementation of the stack is encapsulated from client classes (classes invoking the methods of the *stack* class). Suppose instead of returning the value of the top element, the operation S.Top() returns a pointer to the stack top. Is this a violation of the principle of data encapsulation ? Justify your answer with example uses of the *stack* class by client classes.

   Answer:

   This is a violation of data encapsulation. If S.Top() returns a pointer, then a client class can have code like

   ```
   p = S.Top() ;
   *p = 30;
   ```

   In the second line, the client class is directly manipulating the stack data structure, without invoking the methods of the stack class.

2. As you know, polymorphism is the ability of a single operator/subprogram to denote multiple function definitions depending on the types of the arguments. You are also familiar with the notion of virtual functions in object oriented languages like C++. Virtual functions allow any derived class to redefine a function defined in the base class. Virtual functions are useful for supporting polymorphism in C++. Can you guess why? Explain with examples.

   Answer:

   A virtual function f can be used to support run-time polymorphism where at run time we decide which implementation of f to choose. If there are mulitple derived classes of a single base class with a virtual function f, each derived class may have its own implementation of f. Depending on the class of the object using f, the choice of f's implementation is fixed at run time. For example consider a base class `shape` with derived classes `square` and `circle`.

```
class shape{
  private
      float dimension ; // side for square, radius for circle
  public:
  virtual float compute_area()
  {   ...  // this definition is always overridden
  }
}

class square: public shape{
public:
  float compute_area()
  {   ....  // computing a square's area, invoked for square objects
  }
}

class circle: public shape{
public:
  float compute_area()
  {   ....  // computing a circle's area, invoked for circle objects
  }
}
```

3. A type $t2$ is a subtype of another type $t1$ if a variable of type $t2$ can appear wherever in a program a variable of type $t1$ can appear. In an object oriented program, is a derived class always a subtype of its base class? Explain your answer.

   Answer:

   Yes, if a subclass inherits all the instance variables, instance methods and class methods of its base class. In other words, a derived class is a subtype of its base class if it cannot hide any of the entities of its base class.

4. Trace the evaluation of the following Smalltalk expression:

$$5 \text{ - } 2 + 4 * 3 \text{ !}$$

Answer:

Smalltalk evaluation is left to right with no precedence among the binary operators. The evaluation proceeds as follows:

Send 2 to subtraction method of 5, returns 3
Send 4 to addition method of 3, returns 7
Send 3 to multiplication method of 7, returns 21

D. **Functional Programming** $(4 + 4 + 4 = 12$ marks)

1. In class, we discussed the following ML definition for reversing a list.

```
fun rev(xs) =
        let fun r(nil ,ys) = ys
             |   r(x::xs,ys)= r(xs,x::ys)
in r(xs, nil) end
```

Alternatively, we could also reverse a list using the following function.

```
fun reverse(nil) = nil
  | reverse(x :: xs) = reverse(xs) @ [x]
```

where `x @ y` is the shorthand notation for `append(x, y)`. Analyze the time complexity of both the implementations in terms of n, the length of the list being reversed.

Answer:

The first implementation is $O(n)$ since the function `r` makes a linear scan of the first argument (the list being reversed). For every element of this list, $O(1)$ computation is done.

In the second implementation, again we make a linear scan of the list being reversed. However, for every element of this list, we need to perform an append operation. The append operation is $O(l_1)$ where l_1 is the length of the first list being appended. The time complexity is

$$const * ((n-1) + (n-2) + \ldots + 1))$$

i.e. $O(n^2)$. The first implementation is more efficient.

2. Functional languages like SML allow the user to define recursive datatypes. For example, a binary tree of integers can be defined by the datatype

```
datatype IntBinTree = Empty |
                      Node of (IntBinTree, int, IntBinTree) ;
```

Define a function `postorder` : IntBinTree $\rightarrow$ int list. This function takes an intBinTree as argument, and returns a list of integers. This list will contain the integers in the binary tree; the order of these integers will be as encountered in post-order traversal. Recall that in the post-order traversal of a binary tree, any node is visited after its left and right subtrees.

Answer:

```
fun postorder(Empty) = nil
|    postorder(Node(t1,n,t2)) =
                     append(postorder(t1), append(postorder(t2), [n]));

fun append(nil, ys) = ys
|   append(x::xs,ys) = x :: append(xs,ys);
```

3. SML uses the innermost evaluation strategy for evaluating expressions. However, other functional languages like Haskell support outermost or lazy evaluation, where arguments to a function are evaluated only on demand (refer class notes). This allows definition of infinite data structures. For example we can define the `even` function as below; `even(n)` returns the infinite list of even numbers starting from `n`.

```
even(n) = n :: even(n + 2)
```

In lazy functional languages like Haskell, to search for a given number `x` in a *sorted* infinite list `l` we can simply evaluate the expression:

`member(x, l)`

where `member(x, l)` returns true if and only if integer `x` is in the (potentially infinite) list of integers `l`. Give a definition of the `member` function, such that outermost evaluation of `member(x, l)` terminates for any integer `x`, and any finite/infinite sorted list of integers `l`. Thus `member(4, even(0))` should return true, and `member(5, even(0))` should return false. Use SML syntax.

Answer:

```
fun member(x, nil) = false
  | member(x, y::ys) = if x < y then false
                                  else if x = y then true
                                                else member(x, ys) ;
```

The trick is to stop searching as soon as you encounter an element bigger than x. This takes care of infinite sorted lists.