*CS2403 Programming Languages*

# Homework 1

*Due*: March 22, 2012 in class

1 (40%) As discussed in the class, many language rules are checked by the compiler, and it is possible to bypass the rules using assembly language after compilation. Consider the following C program:

```
#include<stdio.h>
int x=3;
int main(void)
{
    int x=5;
   printf("x = %d\n", x);
  return 0;
}
```

(1) Compile the program and generate its assembly code. (2) Understand the assembly code and modify it to let the program print the global variable x instead of the local variable x.

2 (20%) Using the following grammar, show whether it is possible to generate a parse tree for the statements given. If so, show its leftmost derivation.

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

(1) `A = A * B + C * A`

(2) `A = B + C * (A + B)`

3 (40%) Use the grammar shown at the end (PL Detective) to derive a parse tree for the following program. Is the operator "+" right-associative or left-associative?

```
VAR xyz: INTEGER;
VAR count: INTEGER;
BEGIN
    xyz := 5;
    count := xyz + 1;
END;
```

## PL Detective Grammar

```
<Program>    →  <Block> | <Block> ;
<DeclList>   →  <Decl> | <Decl> ; <DeclList> | ε
<Decl>       →  VAR id : <Type> | TYPE id = <Type> | <ProcDecl>
```

```
<ProcDecl>    →  PROCEDURE id ( <Formals> ) : <Type> = <Block>
                 | PROCEDURE id (<Formals>) = <Block>
<Formals>     →  <FormalList> | ε
<FormalList>  →  <Formal> | <FormalList> ; <Formal>
<Formal>      →  id : <Type>
<Type>        →  INTEGER | <SubrTy> | <ArrayTy> | id | <ProcTy>
<SubrTy>      →  [ Number TO Number]
<ArrayTy>     →  ARRAY <SubrTy> OF <Type>
<ProcTy>      →  PROCEDURE ( <Formals> ) : <Type> | PROCEDURE (<Formals>)
<Block>       →  <DeclList> BEGIN <StmtList> END
<StmtList>    →  <Stmt> | <Stmt> ; <StmtList> | ε
<Stmt>        →  <Assignment> | <Return> | <Block> | <Conditional>
                 | <Iteration> | <Output> | <Expr>
<Assignment>  →  <Expr> := <Expr>
<Return>      →  RETURN <Expr>
<Conditional> →  IF <Expr> THEN <StmtList> ELSE <StmtList> END
<Iteration>   →  WHILE <Expr> DO <StmtList> END
<Output>      →  PRINT ( <Expr> )
<Expr>        →  <Operand> | <Expr> <Operator> <Operand>
<Operand>     →  Number | id | <Operand> [<Expr>]
                 | <Operand>( <Actuals> ) | (<Expr>)
<Operator>    →  + | > | AND
<Actuals>     →  <ActualList> | ε
<ActualList>  →  <Expr> | <ActualList> , <Expr>
```

Notes:

- <Program> is the start symbol of the grammar.
- The symbol ε is empty string.
- This grammar is in BNF, not in EBNF. Particularly, the '[' and ']' are terminals in the language: they do not mean "optional" in EBNF.
- The words in upper case are reserved words of the language (e.g., PROCEDURE and AND)
- <SubrTy> is a subrange type. For example, a variable declared to be of subrange type [1 TO 10] can hold values between 1 and 10 only.
- Numbers include both negative and positive integers.
- id (which are variables or procedure names) are a string of characters (a-zA-Z)