Department of Computer Science
Ashoka University

Programming Language Design and Implementation (PLDI): CS-1319-1

Quiz 4 Solutions

Marks: 100

Date: *December 05, 2023*　　Time: *11:50 - 13:30*

Instructions:

1. The quiz will be physical, on paper, and in the classroom.
2. Write your name and Ashoka ID on the answer-script and additional papers.
3. The quiz comprises three questions (totalling 100 marks) and one bonus question (for 10 marks). Each question has multiple parts with marks shown for each.
4. The test is entirely closed book.
5. Any copy from peers will be dealt with zero tolerance - both to get zero in the question. Consultations or chats with others will lead to zero score for the entire quiz.
6. No question or doubt will be entertained. If you have any query, make your own assumptions, state them clearly in your answer and proceed.
7. Write in clear handwriting and in an unambiguous manner. If TAs have difficulty reading / understanding your answer, they will make assumptions at their best capacity to evaluate. You would not get an opportunity for explanation or rebuttal.

---

1. Write the regular expressions for the following specification of Floating Point Literals using the Flex notation: [5]

   - Every literal has an optional sign (+ / -)
   - Every literal has an optional integral part (comprising zero or more digits), followed by an optional decimal point (.), and then followed by an optional fractional part (comprising zero or more digits)
   - In no literal can the decimal be present and fractional part have 0 digits.
   - In no literal, both the integral and the fractional part may be omitted
   - Valid floating point literals: `1`, `.23`,`0.95`, `123.987`
   - Invalid literals: `47.`, `.`, `97.1.2`

   Ans.

```
FPL     ^[+-]?([0-9]+|[0-9]*\.[0-9]+)
```

   Rubric: 2 points for correct idea. 1 point each for a passing test-case.

2. Write the Flex and Bison specifications for a simple calculator: **[5 + 10 = 15]**

```
stmt -> expr;
expr -> (expr)
expr -> + expr
expr -> - expr
expr -> expr ! expr
expr -> expr + expr
expr -> expr - expr
expr -> expr * expr
expr -> expr / expr
expr -> num
```

   where unary operators (+ -) have the highest precedence and are right associative, followed by exponentiation (!) that is right associative, followed by multiplication (*) and division (/) that are left associative, and finally followed by addition (+) and subtraction (-) that are left associative. Parentheses (( )) are used to override associativity and precedence.

   `num` is an integer numeric literal. The calculator prints the value of `expr` when `stmt -> expr;` is reduced.

   Ans. Comment lines (/*---*/) indicate sections needed in your answer.

   Flex Specifications:

```
%{
    #include "3_tab.h"
%}
/*------------------------------*/
ws      [ \n\t\r]
num     [0-9]+
op      [+\-*/!]
punct   [();]
%%
{ws}    ;
{num}   {   yylval.d_val = atoi(yytext);   return NUM; }
{op}    {   return yytext[0];   }
{punct} {   return yytext[0];   }
.       {   return yywrap();    }
%%
/*------------------------------*/
int yywrap(){   return 1;   }
```

Bison Specifications:

```
%{
    #include <stdio.h>
    #include <math.h>
    extern char *yytext;
    extern int yylex();
    void yyerror(char * s);
%}
/*------------------------------*/
%union{
    double d_val;
}

%type<d_val> expr stmt
%token<d_val> NUM

%left '+' '-'
%left '/' '*'
%right '!'
%right UPLUS UMINUS

%%
stmt:
    expr  ';'                {printf("%lf\n", $1);}
;
expr:
    '(' expr ')'             {$$ = $2;}
|   '+' expr %prec UPLUS     {$$ = +$2;}
|   '-' expr %prec UMINUS    {$$ = -$2;}
|   expr '!' expr            {$$ = pow($1, $3);}
|   expr '/' expr            {
                             if($3 == 0){yyerror("Division by 0"); return 1;}
                             $$ = $1 / $3;
                             }
|   expr '*' expr            {$$ = $1 * $3;}
|   expr '+' expr            {$$ = $1 + $3;}
|   expr '-' expr            {$$ = $1 - $3;}
|   NUM                      {$$ = $1;}
;
%%
/*------------------------------*/
void yyerror(char * s){
    printf("Error: %s on '%s'\n", s, yytext);
}

int main(){
    yyparse();
    return 0;
}
```

Note: Points will not be deducted for the following:

(a) Not having `ws` rule/definition

(b) Using `int, float` instead of `double`

(c) Not doing `"Division by 0"` semantic check.

(d) Not using `yywrap()` in catch-all rule.

(e) Minor syntactic errors (as long as what you were trying to do is fairly clear)

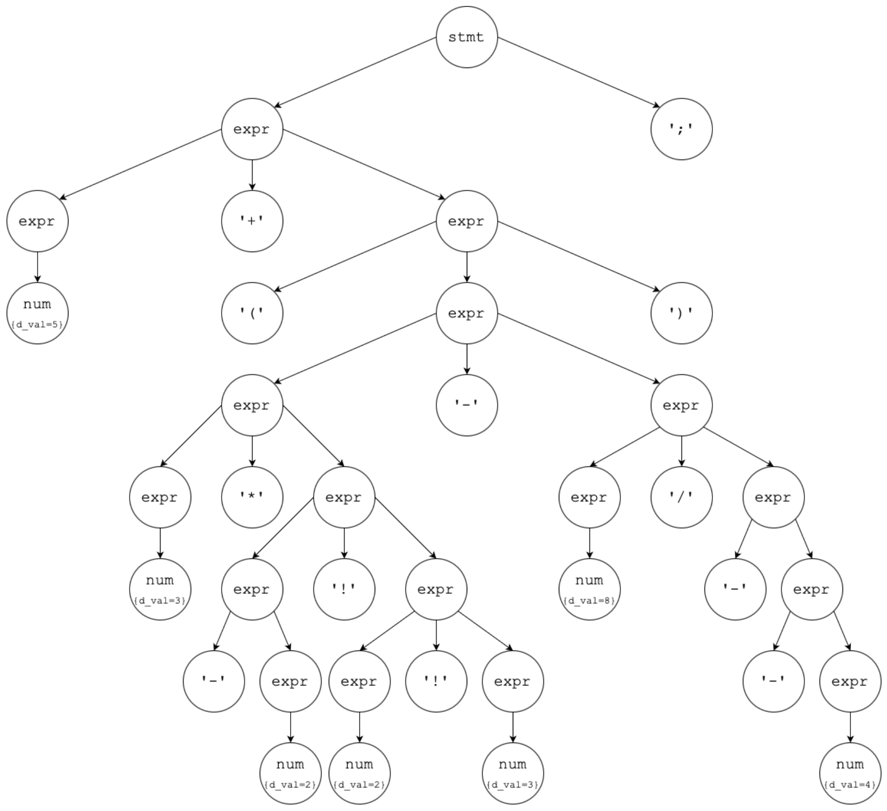
3. Consider the following input for Q2: **[5 + (2 + 1) + 7 = 15]**

```
    5 + (3 * - 2 ! 2 ! 3 - 8 / - - 4) ;
```

Since ; was missing from the question, an answer which states "Syntax Error" for all three sub-parts will also be considered correct. The following answers are for the corrected string.

(a) Draw the parse tree of the input.

Ans. Since grammar is left-recursive, we can only perform bottom-up parsing (unless we remove left recursion). Below is the parse-tree made by given expression.



(b) Fully parenthesize the input expression based on the associativity and precedence of the operators. Manually evaluate the expression.

Ans.
Parenthesization: Either

$$(5 + ((3 * ((-2) ! (2 ! 3))) - (8 / (-(-4)))))$$

or,

$$((5) + (((3) * ((-(2)) ! ((2) ! (3)))) - ((8) / (-(-(4))))))$$

Evaluation: 771

(c) Show the working of the calculator in Q2 using the symbol and value stacks for the input and compute the output. Verify the result with the manual computation in Q3b.

Ans. `expr` has been shortened to `exp`

exp	5

exp → num

exp	3
(	
+	
exp	5

exp → num

exp	2
-	
*	
exp	3
(	
+	
exp	5

exp → num

exp	-2
*	
exp	3
(	
+	
exp	5

exp → −exp

exp	2
!	
exp	-2
*	
exp	3
(	
+	
exp	5

exp → num

exp	3
!	
exp	2
!	
exp	-2
*	
exp	3
(	
+	
exp	5

exp → num

exp	8
!	
exp	-2
*	
exp	3
(	
+	
exp	5

exp → exp ! exp

exp	256
*	
exp	3
(	
+	
exp	5

exp → exp ! exp

exp	768
(	
+	
exp	5

exp → exp * exp

exp	8
-	
exp	768
(	
+	
exp	5

exp → num

exp	4
-	
-	
/	
exp	8
-	
exp	768
(	
+	
exp	5

exp → num

exp	-4
-	
/	
exp	8
-	
exp	768
(	
+	
exp	5

exp → −exp

exp	4
/	
exp	8
-	
exp	768
(	
+	
exp	5

exp → −exp

exp	2
-	
exp	768
(	
+	
exp	5

exp → exp / exp

exp	766
(	
+	
exp	5

exp → exp − exp

)	
exp	766
(	
+	
exp	5

exp	766
+	
exp	5

exp → (exp)

exp	771

exp → exp + exp

;	
exp	771

stmt	771

stmt → exp;

Evaluation: 771

4. Consider the following program:

```c
// IO Library header
int printStr(char *s);
int printInt(int n);
int readInt(int *eP);

int divides(int n, int d) {
    if (n % d == 0)
        return 1;
    else
        return 0;
}

int divisors(int n) {
    int i;
    int sum;

    i = 1;
    sum = 0;
    while (i < n) {
        if (divides(n, i))
            sum = sum + i;
        i++;
    }
    return sum;
}

int perfect(int n) {
    if (divisors(n) == n)
        return 1;
    else
        return 0;
}

int main() {
    int n;

    printStr("Enter the number: ");
    n = readInt(0);

    printInt(n);

    if (perfect(n))
        printStr(" is a perfect number");
    else
        printStr(" is not a perfect number");

    return 0;
}
```

(a) Translate the above program to three address codes:

i. Show the Global Symbol Table with the symbol name, data type, category, and size. Mark appropriate parent / child symbol table pointers to build the tree of symbol tables. [7]
For a type T, use the type expression ptr(T) for T* type.

Ans. Code relevant to global symbol table:

```
// IO Library header
int printStr(char *s);
int printInt(int n);
int readInt(int *eP);

int divides(int n, int d);
int divisors(int n);
int perfect(int n);
int main();
```

ST.glb				Parent: *Null*
Name	**Type**	**Category**	**Size**	**Offset**
printStr	ptr(char) → int	func	0	ST.printStr
printInt	int → int	func	0	ST.printInt
readInt	ptr(int) → int	func	0	ST.readInt
divides	int × int → int	func	0	ST.divides
divisors	int → int	func	0	ST.divisors
perfect	int → int	func	0	ST.perfect
main	void → int	func	0	ST.main

Figure 1: glb Symbol Table

ii. Generate the array of quad codes starting at index 100. [10 + 7 + 3 = 20]

A. For function divisors.

B. For function main.

C. Show the offset entries for constants if needed.

Ans. Quads for divisors:

```
; int divisors(int n)
100: t1 = 1
101: i = t1
102: t2 = 0
103: sum = t2
104: if (i < n) goto 106
105: goto 116
106: param i
107: param n
108: t3 = call divides, 2
109: if (t3 != 0) goto 111
110: goto 113
111: t4 = sum + i
112: sum = t4
113: t5 = i
114: i = i + 1
115: goto 104
116: ret sum
```

Quads for `main`:

```
; int main()
offset _s1 "Enter the number: "
offset _s2 " is a perfect number"
offset _s3 " is not a perfect number"

100: param _s1
101: call printStr, 1
102: t1 = 0
103: param t1
104: n = call readInt, 1
105: param n
106: call printInt, n
107: param n
108: t2 = call perfect, 1
109: if (t2 != 0) goto 111
110: goto 114
111: param _s2
112: call printStr, 1
113: goto 116
114: param _s3
115: call printStr, 1
116: t3 = 0
117: ret t3
```

Offsets for constants:

```
offset _s1 "Enter the number: "
offset _s2 " is a perfect number"
offset _s3 " is not a perfect number"
```

iii. Show the Symbol Table for function `divisors` with the symbol name, data type, category, size, and offset. Mark appropriate parent / child symbol table pointers to build the tree of symbol tables. [5]

Ans.

ST.divisors			Parent: *ST.glb*	
Name	**Type**	**Category**	**Size**	**Offset**
`n`	int	param	4	+8
`i`	int	local	4	-4
`sum`	int	local	4	-8
`t1` ... `t5`	int	temp	4	-12 ... -28

Figure 2: `divisors` Symbol Table

Note: Points will not be deducted for the following:

(a) Having a different number of `temp` variables

(b) Local offset starting at `0`

(c) Param offset starting at `+4`

(b) Peephole optimize the code of function `divisors` and renumber the optimized quads from 100.
[8 + 4 + 3 = 15]

You need to show your output in three stages. First mark the quads for optimization, next renumber the quads, finally clean-up the list of quads.

Here are the typical peephole optimizations for reference.

- def-use and removing dead code

Given Code	**Optimized Code**

`t1 = 3`
`x = t1`
	`x = 3`

- propagating copies and removing dead code

Given Code	**Optimized Code**

`x = a + b`
`y = x`
	`y = a + b`

- folding constants

Given Code	**Optimized Code**
`x = 3 + 1`	`x = 4`

- short-cutting jump-to-jump

Given Code	**Optimized Code**

`[110]: goto 113`
`...`
`[113]: goto 119`
	`[110]: goto 119`

- eliminating jump-over-jump

Given Code	**Optimized Code**

`[110]: if a > b goto 112`
`[111]: goto 120`
`[112]: x = 0`
	`[110]: if a <= b goto 120`
	`[112]: x = 0`

- applying algebraic simplification

Given Code	**Optimized Code**
`x = p + 0`	`x = p`

- applying strength reduction

Given Code	**Optimized Code**
`a = 4 * i`	`a = i << 2`

Ans.

```
; int divisors(int n): OPT. MARKED
100: t1 = 1                         ; deadcode
101: i = 1                          ; def-use
102: t2 = 0                         ; deadcode
103: sum = 0                        ; def-use
104: if (i >= n) goto 116 (106)     ; jump-over-jump: Flipped i < n
105: goto 116                       ; fall-through
106: param i
107: param n
108: t3 = call divides, 2
109: if (t3 == 0) goto 113 (111)    ; jump-over-jump: Flipped t3 != 0
110: goto 113                       ; fall-through
111: t4 = sum + i                   ; deadcode
112: sum = sum + i                  ; def-use
113: t5 = i                         ; deadcode
114: i = i + 1
115: goto 104
116: ret sum
```

```
; int divisors(int n): RE-NUMBERED
100:                                     ; deadcode
101: 100: i = 1                          ; def-use
102:                                     ; deadcode
103: 101: sum = 0                        ; def-use
104: 102: if (i >= n) goto 110 (116)     ; jump-over-jump
105:                                     ; fall-through
106: 103: param i
107: 104: param n
108: 105: t3 = call divides, 2
109: 106: if (t3 == 0) goto 108 (113)    ; jump-over-jump
110:                                     ; fall-through
111:                                     ; deadcode
112: 107: sum = sum + i                  ; def-use
113:                                     ; deadcode
114: 108: i = i + 1
115: 109: goto 102 (104)                 ; compacted
116: 110: ret sum
```

```
; int divisors(int n): CLEANED
100: i = 1
101: sum = 0
102: if (i >= n) goto 110
103: param i
104: param n
105: t3 = call divides, 2
106: if (t3 == 0) goto 108
107: sum = sum + i
108: i = i + 1
109: goto 102
110: ret sum
```

(c) Construct the Control Flow Graph (CFG) for functions `divisors`.

[3 + 10 = 13]

i. Identify the leader quads

ii. Construct the basic blocks and build the CFG.

Ans.

i. Leader Quads:

```
; int divisors(int n)
100: i = 1                    ; leader: rule 1
101: sum = 0
102: if (i >= n) goto 110     ; leader: rule 3
103: param i                  ; leader: rule 2
104: param n
105: t3 = call divides, 2
106: if (t3 == 0) goto 108
107: sum = sum + i            ; leader: rule 2
108: i = i + 1                ; leader: rule 3
109: goto 102
110: ret sum                  ; leader: rule 2, 3
```

ii. **CFG**:

```
; int divisors(int n): CFG
; BLOCK B1
100: i = 1                    ; leader: rule 1
101: sum = 0
   : goto B2

; BLOCK B2
102: if (i >= n) goto B6 (110)    ; leader: rule 3
   : goto B3

; BLOCK B3
103: param i                  ; leader: rule 2
104: param n
105: t3 = call divides, 2
106: if (t3 == 0) goto B5 (108)
   : goto B4

; BLOCK B4
107: sum = sum + i            ; leader: rule 2
   : goto B5

; BLOCK B5
108: i = i + 1                ; leader: rule 3
109: goto B2 (102)

; BLOCK B6
110: ret sum                  ; leader: rule 2, 3
```

(d) Answer the following questions in the context of liveness of variables in the CFG of `divisors` as constructed in Q4c. **[2 + 2 + 1 = 5]**

i. Mark the live variables at the entry and exit of every basic block.

ii. Using the information of which variables are live at entry and at exit of a block, estimate which variables are live at every point (between any two consecutive statements) within a block.
You do not need to show the live variables at every program point. Just mark if a variable becomes live on the execution of a quad or becomes dead after it. You do not need to show anything for the quads that do not change the liveness of any variable.

iii. Based on the information of live variables thus computed, determine the minimum number of registers needed for function `divisors` if we do not want to spill (keep copy of) any variable to memory. Justify your answer.

Ans.

i., ii. and iii. **Live Variables at Entry & Exit of Basic Blocks:**

```
; int divisors(int n): Live Variables

; BLOCK B1
; Live variables = {n}
100: i = 1
101: sum = 0
; Live variables = {i, n, sum}
   : goto B2

; BLOCK B2
; Live variables = {i, sum}
102: if (i >= n) goto B6
; Live variables = {i, n, sum}
   : goto B3

; BLOCK B3
; Live variables = {i, n, sum}
103: param i
104: param n
105: t3 = call divides, 2    ; t3 is live
106: if (t3 == 0) goto B5    ; t3 is live
; Live variables = {i, n, sum}
   : goto B4

; BLOCK B4
; Live variables = {i, n, sum}
107: sum = sum + i
; Live variables = {i, n, sum}
   : goto B5

; BLOCK B5
; Live variables = {i, n, sum}
108: i = i + 1
; Live variables = {i, n, sum}
109: goto B2 (102)

; BLOCK B6
; Live variables = {sum}
110: ret sum
```

Live Variables at every program point

Shown above as comment

Minimum number of registers needed

In Block B3 between statements 105 and 106, four variable, viz. `i, n, sum, t3` are live together. So we need 4 registers at the minimum.

5. **Bonus:** Write a C function to convert the `yytext` of the Floating Point Literal in Q1 to an equivalent `double` value without using the `atof` function of C standard library. **[10]**

Ans.

```c
double string_to_double(char * s, int n){
    int i = 0;
    double integral = 0, fractional = 0, num = 0;
    // Calculate the integral part
    while(i < n && s[i] != '.'){
        integral = integral * 10 + (s[i] - '0');
        i++;
    }
    i = n - 1;
    //Calculate the fractional part
    while(i >= 0 && s[i] != '.'){
        fractional = fractional/10 + (double)(s[i] - '0')/10;
        i--;
    }
    // Add the two to get resulting double
    num = integral + fractional;

    return num;
}
```