Final\_Exam

Hyunki Lee

Panther# 002-34-4677

1.

The numeric type can be an example that a simple assignment statement is legal in C++ but not in java. For example, if A is integer, and B is floating number, then we can see the result in C++ since the converting between integer and floating is valid. However, the simple assignment does not work in Java thus the assignment is illegal in Java.

2.

Implicit type conversion: Implicit type conversion is that data is converted without losing the values inside the variable. Example) int a; long b = 3; a = b;

Benefit:

Writability: The implicit conversion is easy to implement.

Drawbacks:

Readability: The implicit conversion reduces the readability.

Reliability: The implicit conversion occurs some errors that

unexpected.

Cost: The implicit conversion must work all the time thus it takes

processing time to convert data type into the compiler required form

Explicit type conversion: Explicit type conversion is called a ‘cast’. The user intends to make a conversion and that the user is aware that data losing might occur. It is possible to fail runtime because of the cast. Example) float b = 3.3; int a = (int)b – 2;

Benefit:

Readability : Explicit conversion is more readable because, we can

easily see the type conversion.

Cost: explicit conversion is cheaper than implicit conversion.

Explicit conversion is easier to debug.

Drawbacks:

Writability: Explicit conversion is not convenient. The writer needs

to consider data losing.

Reliability: explicit conversion might occur data losing or raise

exceptions

3.

1) a \* b -1 + c

(a \* b)1 -1 + c

((a \* b)1 -1)2 + c

(((a\*b)1-1)2 + c)3

2) a \* (b-1) / c % d

a \* (b-1)1 / c % d

(a \* (b-1)1 )2/ c % d

(a \* (b-1)1 )2/ (c % d)3

((a \* (b-1)1)2/ (c % d)3)4

3) (a – b) / c & (d \* e / a – 3)

(a – b)1 / c & (d \* e / a – 3)

(a – b)1 / c & ((d \* e)2 / a – 3)

(a – b)1 / c & ((d \* e)2 / (a – 3)3)

(a – b)1 / c & ((d \* e)2 / (a – 3)3)4

((a – b)1 / (c & ((d \* e)2 / (a – 3)3)4)5

(((a – b)1 / (c & ((d \* e)2 / (a – 3)3)4)5)6

4) ( a + b <= c ) \* ( d > b - e )

( (a + b)1 <= c ) \* ( d > b - e )

( (a + b)1 <= c )2 \* ( d > b - e )

( (a + b)1 <= c )2 \* ( d > (b – e)3 )

( (a + b)1 <= c )2 \* ( d > (b – e)3 )4

(( (a + b)1 <= c )2 \* ( d > (b – e)3 )4)5

5) -a || c = d && e

(-a)1 || c = d && e

(-a)1 || c = (d && e)2

((-a)1 || c)3 = (d && e)2

(((-a)1 || c)3 = (d && e)2)4

6) a > b ˜| c || d <= 17

a > b ˜| c || (d <= 17)1

(a > b)2 ˜| c || (d <= 17)1

((a > b)2 ˜| c)3 || (d <= 17)1

(((a > b)2 ˜| c)3 || (d <= 17)1)4

7) -a + b

-(a + b)1

(-(a + b)1)2

8) a + b \* c + d

a + (b \* c)1 + d

(a + (b \* c)1)2 + d

((a + (b \* c)1 )2+ d)3

9) E = ++(a++)

E = ++(a++)1

E = (++(a++)1)2

(E = (++(a++)1)2)3

4.

1) a \* b -1 + c

(5 \* 7)1 -1 + c

(35 -1)2 + c

(34+ 11)3

45 (can’t represent to use 5bit)

(01) 01101

2) a \* (b-1) / c % d

a \* (7-1)1 / c % d

(5 \* 6 )2/ 11 % -13

30 / -2

-15

10001

3) (a – b) / c & (d \* e / a – 3)

(5 – 7)1 / c & (d \* e / a – 3)

-2 / c & ((-13 \* -2)2 / a – 3)

-2 / c & (26 / (5 – 3)3)

-2 / c & (26 / 2)4

(-2 / (11 & 13)5

-2/(01011 & 01101)

-2/(01001)

-2/9

11110/01001

Remainder 🡺 011

4) ( a + b <= c ) \* ( d > b - e )

( (5 + 7)1 <= c ) \* ( d > b - e )

( 12 <= 11 )2 \* ( d > b - e )

(12<=11) \* ( d > 9 )

(12<=11) \* ( -13 > 9 )

0(false) \* 0(false)

00000

5) -a || c = d && e

-5 || c = d && e

-5 || c = (-13 && -2)2

-5 || 11 = -13 && -2

1(true) = 1(true)

00001

6) a > b ˜| c || d <= 17

a > b ˜| c || (-13 <= 17)1

(5 > 7)2 ˜| c || 1(true)

(0(false) ˜| 11)3 || 1

00000 ˜| 01011 || 1

01011 || 1

11 || 1

1(true)

00001

7) -a + b

-(5 + 7)1

(-12)2

10100

8) a + b \* c + d

a + (7 \* 11)1 + d

(5 + 77)2 + d

(82-13)3

69(can’t represent to use 5bit)

(010) 00101

9) E = ++(a++)

E = ++(5++)1

E = (++5)2

-2 = ++5

0(false)

00000

5.

V = { Stmt, Postfix, Prefix, unary\_op, binary\_op, bitwise\_op, incre\_op,

decre\_op}

E = { variables, =, /= }

R = [

Stmt => Postfix incre\_op | Postfix decre\_op

Stmt => incre\_op Prefix | decre\_op Prefix

Stmt => unary\_op variables

Stmt => Stmt | Stmt Stmt | Stmt = Stmt

Stmt => variables binary\_op variables | variables bitwise\_op variables

Postfix => variables

Prefix => variables

unary\_op => + | - | ! | \*

binary\_op => “||” | && | / | % | >= | <= | + | - | ! | \* | > | <

bitwise\_op => & | “˜|”

incre\_op => ++

decre\_op => --

]

S = Stmt

6.

1) a \* b -1 + c

(((a\*b)1-1)2 + c)3

((a.multiplication(b)).minus(1)).plus(c)

2) a \* (b-1) / c % d

((a \* (b-1)1)2/ (c % d)3)4

a.multiplication(b.minus(1)).division(c.modulus(d))

3) (a – b) / c & (d \* e / a – 3)

(((a – b)1 / (c & ((d \* e)2 / (a – 3)3)4)5)6

(a.minus(b)).division(c.bitwiseAND(d.multiplication.(e)).division(a.minus(3)))

4) ( a + b <= c ) \* ( d > b - e )

(( (a + b)1 <= c )2 \* ( d > (b – e)3 )4)5

((a.plus(b)).comparision(c)).multiplication(d.comparision(b.minus(e)))

5) -a || c = d && e

(((-a)1 || c)3 = (d && e)2)4

((-a).logicalOR(c)).equal((d.logicalAND(e)))

6) a > b ˜| c || d <= 17

(((a > b)2 ˜| c)3 || (d <= 17)1)4

((a.comparision(b)).bitwiseOR(c)).logicalOR(d.comparision(17))

7) -a + b

(-(a + b)1)2

-(a.plus(b))

8) a + b \* c + d

((a + (b \* c)1 )2+ d)3

((b.mutiplication(c)).plus(a)).plus(d)

9) E = ++(a++)

(E = (++(a++)1)2)3

E.equal(increment.(a.increment))

No. There is not a need express to represent precedence. We can express each expression using function calls. Example) a + b \* c + d. we need to multiply b and c then add a and add d. We can express like

((b.mutiplication(c)).plus(a)).plus(d) 🡺 We can see the order of processing.

7.

Copy repl link

<https://repl.it/@todok4636/PLCFinalQ7>

Inviting link

https://repl.it/join/vypzsmxv-todok4636

1) a \* b -1 + c

(((a\*b)1-1)2 + c)3

((a.multiplication(b)).minus(1)).plus(c)

    void stmt(){

      LEFT\_PAREN();

      LEFT\_PAREN();

      lex();

      dot();

      if(nextToken != multiplication)

        error();

      else{

        LEFT\_PAREN();

        lex();

        RIGHT\_PAREN();

        RIGHT\_PAREN();

        dot();

        if(nextToken != minus)

          error();

        else{

          LEFT\_PAREN();

          lex();

          RIGHT\_PAREN();

          RIGHT\_PAREN();

          dot();

          if(nextToken != plus)

            error();

          else{

            LEFT\_PAREN();

            lex();

            RIGHT\_PAREN();

          }

        }

      }

    }

2) a \* (b-1) / c % d

((a \* (b-1)1)2/ (c % d)3)4

a.multiplication(b.minus(1)).division(c.modulus(d))

    void stmt(){

      lex();

dot();

      if(nextToken != multiplication)

        error();

      else{

        LEFT\_PAREN();

        lex();

        dot();

        if(nextToken != minus)

          error();

        else{

          LEFT\_PAREN();

          lex();

          RIGHT\_PAREN();

          RIGHT\_PAREN();

          dot();

          if(nextToken != division)

            error();

          else{

            RIGHT\_PAREN();

            lex();

            dot();

            if(nextToken != modulus)

              error();

            else{

              RIGHT\_PAREN();

              lex();

              LEFT\_PAREN();

              LEFT\_PAREN();

            }

          }

        }

      }

    }

3) (a – b) / c & (d \* e / a – 3)

(((a – b)1 / (c & ((d \* e)2 / (a – 3)3)4)5)6

(a.minus(b)).division(c.bitwiseAND(d.multiplication.(e)).division(a.minus(3)))

void stmt(){

RIGHT\_PAREN();

lex();

dot();

if(nextToken != minus)

error();

else{

RIGHT\_PAREN();

lex();

LEFT\_PAREN();

LEFT\_PAREN();

dot();

if(nextToken != division)

error();

else{

RIGHT\_PAREN();

lex();

dot();

if(nextToken != bitwiseAND)

error();

else{

RIGHT\_PAREN();

lex();

dot();

if(nextToken != multiplication)

error();

else{

RIGHT\_PAREN();

lex();

LEFT\_PAREN();

LEFT\_PAREN();

dot();

if(nextToken != division)

error();

else{

RIGHT\_PAREN();

lex();

dot();

if(nextToken != minus)

error();

else{

LEFT\_PAREN();

lex();

RIGHT\_PAREN();

RIGHT\_PAREN();

RIGHT\_PAREN();

}

}

}

}

}

}

}

4) ( a + b <= c ) \* ( d > b - e )

(( (a + b)1 <= c )2 \* ( d > (b – e)3 )4)5

((a.plus(b)).comparision(c)).multiplication(d.comparision(b.minus(e)))

void stmt(){

LEFT\_PAREN();

LEFT\_PAREN();

lex();

dot();

if(nextToken != plus)

error();

else{

LEFT\_PAREN();

lex();

RIGHT\_PAREN();

RIGHT\_PAREN();

dot();

if(nextToken != comparision)

error();

else{

LEFT\_PAREN();

lex();

RIGHT\_PAREN();

RIGHT\_PAREN();

dot();

if(nextToken != multiplication)

error();

else{

LEFT\_PAREN();

lex();

dot();

if(nextToken != comparision)

error();

else{

LEFT\_PAREN();

lex();

dot();

LEFT\_PAREN();

lex();

dot();

if(nextToken != minus)

error();

else{

LEFT\_PAREN();

lex();

RIGHT\_PAREN();

RIGHT\_PAREN();

RIGHT\_PAREN();

}

}

}

}

}

}

5) -a || c = d && e

(((-a)1 || c)3 = (d && e)2)4

((-a).logicalOR(c)).equal((d.logicalAND(e)))

void stmt(){

LEFT\_PAREN();

LEFT\_PAREN();

unary\_op().minus();

lex();

RIGHT\_PAREN();

dot();

if(nextToken != logicalOR)

error();

else{

LEFT\_PAREN();

lex();

RIGHT\_PAREN();

RIGHT\_PAREN();

dot();

if(nextToken != equal)

error();

else{

LEFT\_PAREN();

LEFT\_PAREN();

lex();

dot();

if(nextToken != logicalAND)

error();

else{

LEFT\_PAREN();

lex();

RIGHT\_PAREN();

RIGHT\_PAREN();

RIGHT\_PAREN();

}

}

}

}

6) a > b ˜| c || d <= 17

(((a > b)2 ˜| c)3 || (d <= 17)1)4

((a.comparision(b)).bitwiseOR(c)).logicalOR(d.comparision(17))

void stmt(){

LEFT\_PAREN();

LEFT\_PAREN();

lex();

dot();

if(nextToken != comparision)

error();

else{

LEFT\_PAREN();

lex();

RIGHT\_PAREN();

RIGHT\_PAREN();

dot();

if(nextToken != bitwiseOR)

error();

else{

LEFT\_PAREN();

lex();

RIGHT\_PAREN();

RIGHT\_PAREN();

dot();

if(nextToken != logicalOR)

error();

else{

LEFT\_PAREN();

lex();

dot();

if(nextToken != comparision)

error();

else{

LEFT\_PAREN();

lex();

RIGHT\_PAREN();

RIGHT\_PAREN();

}

}

}

}

}

7) -a + b

(-(a + b)1)2

-(a.plus(b))

void stmt(){

unary\_op().minus();

LEFT\_PAREN();

lex();

dot();

if(nextToken != plus)

error();

else{

LEFT\_PAREN();

lex();

RIGHT\_PAREN();

RIGHT\_PAREN();

}

}

8) a + b \* c + d

((a + (b \* c)1 )2+ d)3

((b.mutiplication(c)).plus(a)).plus(d)

void stmt(){

LEFT\_PAREN();

LEFT\_PAREN();

lex();

dot();

if(nextToken != multiplication)

error();

else{

LEFT\_PAREN();

lex();

RIGHT\_PAREN();

RIGHT\_PAREN();

dot();

if(nextToken != plus)

error();

else{

LEFT\_PAREN();

lex();

RIGHT\_PAREN();

RIGHT\_PAREN();

dot();

if(nextToken != plus)

error();

else{

LEFT\_PAREN();

lex();

RIGHT\_PAREN();

}

}

}

}

9) E = ++(a++)

(E = (++(a++)1)2)3

E.equal(increment.(a.increment))

void stmt(){

lex();

dot();

if(nextToken != equal)

error();

else{

LEFT\_PAREN();

if(nextToken != increment)

error();

else{

dot();

LEFT\_PAREN();

lex();

dot();

if(nextToken != increment)

error();

else{

RIGHT\_PAREN();

RIGHT\_PAREN();

}

}

}

}

8.

Copy repl link

https://repl.it/@todok4636/PLCFinalQ8HyunkiLee

Inviting link

https://repl.it/join/ulaptgxb-todok4636

I will track variables and operators as tokens. When we tokenize the variables, we do not care about variables’ type or value.

Source Code

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<ctype.h>

//define each tokens as unique number

#define PLUS 1

#define ASSIGNMENT 2

#define MINUS 3

#define DIVISION 4

#define MULTI 5

#define MODULO 6

#define NOT 7

#define OPEN\_FUNC 8

#define CLOSE\_FUNC 9

#define INCRE 10

#define DECRE 11

#define AND 12

#define OR 13

#define LEFT 14

#define RIGHT 15

#define LEFT\_EQUAL 16

#define RIGHT\_EQUAL 17

#define XOR 18

#define a1 19

#define b1 20

#define c1 21

#define d1 22

#define e1 23

#define SPACE 24

#define IDENTIFIER 25

#define DIGIT 26

#define FLOATING 27

#define TOTAL 28

int numOfToken = 0;

//Create struct frame

struct list{

  char pick[35];

  int token;

};

//Type struct elements

struct list reference[TOTAL] = {

  {"+", PLUS},

  {"=", ASSIGNMENT},

  {"-", MINUS},

  {"/", DIVISION},

  {"\*", MULTI},

  {"%", MODULO},

  {"!", NOT},

  {"(", OPEN\_FUNC},

  {")", CLOSE\_FUNC},

  {"++", INCRE},

  {"--", DECRE},

  {"&&", AND},

  {"||", OR},

  {">", LEFT},

  {"<", RIGHT},

  {">=", LEFT\_EQUAL},

  {"<=", RIGHT\_EQUAL},

  {"~|", XOR},

  {"a", a1},

  {"b", b1},

  {"c", c1},

  {"d", d1},

  {"e", e1}

};

// tokens category that where the tokens belong to (This is related to reference struct)

char tokenCategory[TOTAL+1][50]={

" ",

"PLUS(4rd precedence)",

"ASSIGNMENT(Lowest precedence)",

"MINUS(4th precedence)",

"DIVISION(7th precedence)",

"MULTI(3rd precedence)",

"MODULO(5th precedence)",

"NOT(7th precedence)",

"OPEN\_FUNC",

"CLOSE\_FUNC",

"INCREMENT(1st precedence)",

"DECREMENT(1st precedence)",

"AND(3rd precedence)",

"OR(8th precedence)",

"LEFT(6th precedence)",

"RIGHT(6th precedence)",

"LEFT\_EQUAL(3rd precedence)",

"RIGHT\_EQUAL(4th precedence)",

"XOR(8th precedence)",

"5",

"7",

"11",

"-13",

"-2",

"SPACE",

"VARIABLES",

"DIGIT",

"FLOATING"

};

//Finding tokens

int parsing(char lex[]){

  int i;

  for (i = 0; i < TOTAL; i++){

    if(strcmp(lex,reference[i].pick) == 0){

      return reference[i].token;

    }

  }

  return IDENTIFIER;

}

//function for printing result

void printResult(int num, char temp[]){

  printf("\n");

  printf("\t\t\t\t%d\t\t\t%s is %s\n",num,temp,tokenCategory[num]);

}

//function for lexical analyzer

void lexi(char temp[], int tempLength){

  int i,j,k;

  int line = 2;

  char c,next;

  char lex[30];

  char z[300];

  //Tokenizing.

  for(i=0; i < tempLength;){

    c = temp[i];

    for(j=0; j<10; j++){

      z[j]='\0';

    }

    j=0;

    z[j++]=temp[i];

    z[j]='\0';

    //Using switch-case to distinguish each token

    switch(c){

      case' ':

        i++;

        printf("\n");

        break;

      case'\t':

        i++;

        printf("\n");

        break;

      case '-':

        next = temp[++i];

        if(next=='-'){

          i++;

          printResult(DECRE,z);

          break;

        }else{

          i++;

          printResult(MINUS,z);

          break;

        }

      case '+':

        next = temp[++i];

        if(next=='+'){

          i++;

          printResult(INCRE,z);

          break;

        }else{

          i++;

          printResult(PLUS,z);

          break;

        }

      case '>':

        next = temp[++i];

        if(next=='='){

          i++;

          printResult(LEFT\_EQUAL,z);

          break;

        }else{

          i++;

          printResult(LEFT,z);

          break;

        }

      case '<':

        next = temp[++i];

        if(next=='='){

          i++;

          printResult(RIGHT\_EQUAL,z);

          break;

        }else{

          i++;

          printResult(RIGHT,z);

          break;

        }

      case '~':

        next = temp[++i];

        if(next=='|'){

          i++;

          printResult(XOR,z);

          break;

        }

      case '=':

        i++;

        printResult(ASSIGNMENT,z);

        break;

      case '%':

        i++;

        printResult(MODULO,z);

        break;

      case '/':

        i++;

        printResult(DIVISION,z);

        break;

      case '\*':

        i++;

        printResult(MULTI,z);

        break;

      case '!':

        i++;

        printResult(NOT,z);

        break;

      case '(':

        i++;

        printResult( OPEN\_FUNC,z);

        break;

      case ')':

        i++;

        printResult(CLOSE\_FUNC,z);

        break;

      case '&':

        i++;

        printResult(AND,z);

        break;

      case '|':

        i++;

        printResult(OR,z);

        break;

      //Set the default tokens. The two categories are alphabet and digits

      default:

        if(isalpha(temp[i])){

          k =0;

          while(isalpha(temp[i])){

            lex[k++] = temp[i++];

          }

          lex[k]='\0';

          printResult(parsing(lex),lex);

          break;

        } if(isdigit(temp[0])){

          printf("\n");

        }

          if(isdigit(temp[i])){

            if(isalpha(temp[i+1])){

              printf("\n");

            }

            k = 0;

            while(isdigit(temp[i])){

              lex[k++] = temp[i++];

            }

            if(temp[i] !='.'){

              lex[k] = '\0';

              printResult(DIGIT,lex);

              break;

            }

            //Floating number conditions.

            else if(temp[i]=='.' && isdigit(temp[i+1])){

              int check=0;

              lex[k++]='.';

              i++;

              while(isdigit(temp[i])){

                lex[k++] = temp[i++];

              }

              while(isdigit(temp[i])){

                if(check==0)

                lex[k++] = temp[i];

                i++;

              }

              if(check==1){

                break;

              }

              lex[k] = '\0';

              printResult(FLOATING,lex);

              break;

            }

          }

        else if(temp[i]=='\n'){

          i++;

          if(temp[i+1] != '\n'){

            printf("\n\nLine No.=%d\n",line++);

            printf("\n");

          }

        }

        else if(temp[i]=='\t' || temp[i]==' '){

          i++;

        }

        else{

          i++;

        }

    }

  }

  for(i=0; i<10;i++){

    z[i]='\0';

  }

}

int main(){

  //ready to read file, create file pointer.

  FILE \*fp;

  int i=0;

  int f;

  char temp[300];

  int tempLength;

  char g[30];

  printf("Test2 Question1\n");

  printf("\n");

  // open file

  fp = fopen("input.txt","r");

  // print an error when file does not exit

  if(fp == NULL){

    printf("Need a text file, the name should be 'input.txt'");

    printf("\n");

  }

  // check each character until end of file

  while((f = getc(fp)) != EOF){

    temp[i++] = f;

  }

  tempLength = i;

  //close file

  fclose(fp);

  printf("\nLine No.\t\t\tToken ID\t\tExplain\n");

  printf("Line No.=1\n");

  printf("a=5, b=7, c=11, d=-13, e=-2");

  lexi(temp, tempLength);

  return 0;

}

9.

a>b>c in math logic. We compare a>b and b>a. Otherwise, in c, the logic will be evaluated from left to right. If a>b is true, then return ‘true’ which is 1. After that the we compare 1>c whether it is true or false. Thus, even though some examples are true in math logic, it is possible to be false in c.

10.

Copy repl link

https://repl.it/@todok4636/PLCFinalQ10HyunkiLee

Inviting link

https://repl.it/join/gflodgmc-todok4636

Source code

#include <stdio.h>

int fun (int \*k){

\*k += 4;

return 3 \* (\*k) -1;

}

int main(void) {

int i = 10;

int j = 10;

int sum1, sum2;

sum1 = (i/j) + fun(&j);

sum2 = fun(&i) + (i/j);

printf("%d\n",sum1);

printf("%d",sum2);

return 0;

}

Result:

sum1: 42

sum2: 42

In sum1, the (i/j) = 1 and fun(&j) is 41(3\*14-1). Thus (i/j) + fun(&j) = 42

In sum2, the fun(&i) = 41(3\*14-1) and (i/j) = 1. Thus fun(&i) + (i/j) = 42

This is related to ‘pointer’ in C. Even though we call fun() and use j or i as a variable, the value of original i and j will not be changed.