

Compilers summer 2009  
Project Part 3  
**Alan Ritacco**

**Contents**

<b>1. Introduction.....</b>	<b>2</b>
<b>2. The Scanner.....</b>	<b>3</b>
<b>2.1 The Tokens .....</b>	<b>3</b>
<b>2.2 The Lex File .....</b>	<b>4</b>
<b>2.3 Inputs .....</b>	<b>5</b>
<b>2.4 Outputs .....</b>	<b>11</b>
<b>3. The Parser.....</b>	<b>12</b>
<b>3.1 The Grammar.....</b>	<b>13</b>
<b>3.2 The Lex and Yacc files.....</b>	<b>15</b>
<b>3.3 Inputs and Output.....</b>	<b>18</b>
<b>4. Abstract Syntax Tree.....</b>	<b>35</b>
<b>4.1 AST Lex file update. ....</b>	<b>36</b>
<b>4.2 AST Yacc file update. ....</b>	<b>40</b>
<b>4.3 Input and Output.....</b>	<b>46</b>
Bibliography .....	53

## 1. Introduction

This project is the creation and testing towards the formation of a compiler for the programming language called Javalet, which is a subset of the Java<sup>(1)</sup> programming language. In project phase I we have created a scanner written using the fast lexical analyzer (Flex<sup>(2)</sup>) parlance, which is based off of Lex<sup>(3)</sup>. We are using Flex to read the symbols, based off of the tokens listed in section 2.1, to lexicographically scan our Flex sources shown in 2.2. The information for part one of this project can be found by browsing to the following URL:

<http://web.cs.wpi.edu/~kal/courses/compilers/project/projectlexernew1.doc>, We have extended the project to include part II of the requirements, shown in section 3 below. We have extended the simple scanner to include logical output tokens in our Flex file, and created a parser from these tokens using Yacc<sup>(4)</sup>. We have implemented a Yacc file to read the language as defined by project 2. The information for part two of this project can be found by browsing to the following URL: <http://web.cs.wpi.edu/~kal/courses/compilers/project/projectparser.doc>, the complete project description can be found by browsing to the following URL <http://web.cs.wpi.edu/~kal/courses/compilers/project/>.

## 2. The Scanner

The scanner portion of this project has been implemented to “scan” for valid tokens from the Javalet language. The tokens for the Javalet language, listed below, have been created in a scanner file and implemented using Flex’s constructs. When running Flex against this scanner file the Flex program output will be a C program which can then be compiled to a binary. We use this binary to scan against our Javalet language file(s) and then be used as our lexical scanner for Javalet. The tokens for the Javalet language are listed below in section 2.1. We have used Flex version 2.5.35, and GCC<sup>(5)</sup> version 4.3 for the lexicographical and compilers respectively. A GNU<sup>(6)</sup> Makefile<sup>(7)</sup> has been created, for convenience, to help with the creation of the Flex output files and the resulting scanner binary file.

### 2.1 The Tokens

The following are the tokens as given for phase I of the project as taken from the class assignment for project phase I.

#### Type

"void", "int", "real"

#### Logical Operators

"!", "||", "&&", "!=", "==", "<", ">", "<=", ">="

#### Numerical Operators

"+", "-", "\*", "/", "=",

#### Punctuation

"{", "}", "(", ")", ",", ";"

#### Keywords

"if", "else", "while", "do", "for"

#### Names

Letter (Letter | Digit | ) \* where a Letter is either an uppercase or lowercase letter and Digit is one of the digits from 0-9. There cannot be 2 consecutive underscores.

#### Integers

Sequences of 1 or more digits

#### Reals

Plus or minus followed by a number of digits followed by a dot ".", followed by a number of digits. Either the sequence before the dot can be null or the sequence after the dot can be null, but not both.



## 2.3 Inputs

The testing files used are number input 1-4. The Input files 1-3 are those as taken from the test cases in project phase I. Test case input 4 includes test for the following (not included in tests 1-3): double underscore input, upper case input, OR operator input, do while input, the c ++ operator input, quote's " ", escape sequence \n, and a positive real in the input. Input not recognized by the scanner at this point is ignored and outputted.

----- **Input 1** -----

```
void input_a() {
    integer a, bb, xyz, b3, c, p, q;
    real b;
    a = b3;
    b = -2.5;
    xyz = 2 + a + bb + c - p / q;
    a = xyz * ( p + q );
    p = a - xyz - p;
}
```

----- **Output for Input1** -----

```
Identifier, void
Names, input_a
Punctuation, OPEN_PARAN
Punctuation, CLOSE_PARAN
Punctuation, OPEN_BRACE
Names, integer
Names, a
Punctuation, COMMA
Names, bb
Punctuation, COMMA
Names, xyz
Punctuation, COMMA
Names, b3
Punctuation, COMMA
Names, c
Punctuation, COMMA
Names, p
Punctuation, COMMA
Names, q
Punctuation, SEMICOLON
Identifier, real
Names, b
Punctuation, SEMICOLON
Names, a
Numerical_Operator, EQUAL
Names, b3
Punctuation, SEMICOLON
Names, b
```

Numerical\_Operator, EQUAL  
Real, -2.5  
Punctuation, SEMICOLON  
Names, xyz  
Numerical\_Operator, EQUAL  
Integer, 2  
Numerical\_Operator, PLUS  
Names, a  
Numerical\_Operator, PLUS  
Names, bb  
Numerical\_Operator, PLUS  
Names, c  
Numerical\_Operator, MINUS  
Names, p  
Numerical\_Operator, DIVIDE  
Names, q  
Punctuation, SEMICOLON  
Names, a  
Numerical\_Operator, EQUAL  
Names, xyz  
Numerical\_Operator, MULTIPLY  
Punctuation, OPEN\_PARAN  
Names, p  
Numerical\_Operator, PLUS  
Names, q  
Punctuation, CLOSE\_PARAN  
Punctuation, SEMICOLON  
Names, p  
Numerical\_Operator, EQUAL  
Names, a  
Numerical\_Operator, MINUS  
Names, xyz  
Numerical\_Operator, MINUS  
Names, p  
Punctuation, SEMICOLON  
Punctuation, CLOSE\_BRACE

-----**Input 2**-----

```
void input_b() {  
    if ( i > j )  
        i = i + j;  
    else if ( i < j )  
        i = 1;  
}
```

----- **Output for input 2** -----

Identifier, void  
Names, input\_b  
Punctuation, OPEN\_PARAN  
Punctuation, CLOSE\_PARAN  
Punctuation, OPEN\_BRACE  
Keyword, IF  
Punctuation, OPEN\_PARAN  
Names, i  
Logical\_Operator, GREATER\_THAN  
Names, j  
Punctuation, CLOSE\_PARAN  
Names, i  
Numerical\_Operator, EQUAL  
Names, i  
Numerical\_Operator, PLUS  
Names, j  
Punctuation, SEMICOLON  
Keyword, ELSE  
Keyword, IF  
Punctuation, OPEN\_PARAN  
Names, i  
Logical\_Operator, LESS\_THAN  
Names, j  
Punctuation, CLOSE\_PARAN  
Names, i  
Numerical\_Operator, EQUAL  
Integer, 1  
Punctuation, SEMICOLON  
Punctuation, CLOSE\_BRACE

### -----Input 3-----

```
void input_c() {  
    while ( i < j && j < k ) {  
        k = k + 1;  
        while ( i == j )  
            i = i + 2;  
    }  
}
```

#### ----- Output for Input 3 -----

Identifier, void  
Names, input\_c  
Punctuation, OPEN\_PARAN  
Punctuation, CLOSE\_PARAN  
Punctuation, OPEN\_BRACE  
Keyword, WHILE  
Punctuation, OPEN\_PARAN  
Names, i  
Logical\_Operator, LESS\_THAN  
Names, j  
Logical\_Operator, AND  
Names, j  
Logical\_Operator, LESS\_THAN  
Names, k  
Punctuation, CLOSE\_PARAN  
Punctuation, OPEN\_BRACE  
Names, k  
Numerical\_Operator, EQUAL  
Names, k  
Numerical\_Operator, PLUS  
Integer, 1  
Punctuation, SEMICOLON  
Keyword, WHILE  
Punctuation, OPEN\_PARAN  
Names, i  
Logical\_Operator, EQUAL  
Names, j  
Punctuation, CLOSE\_PARAN  
Names, i  
Numerical\_Operator, EQUAL  
Names, i  
Numerical\_Operator, PLUS  
Integer, 2  
Punctuation, SEMICOLON  
Punctuation, CLOSE\_BRACE  
Punctuation, CLOSE\_BRACE

----- **Input 4: An Example of your own** -----

```
int test(int x,y)
{
    int X,Y__Z;
    X=+4.5;
    Y__Z=-1.2;

    if (x || z)
        x=y;
    else
        x=2;
    x=5, z=5;
    do while(x<6)
    {
        x++;
    }
}
```

---- **Output for input 4** ----

Identifier, int  
Names, test  
Punctuation, OPEN\_PARAN  
Identifier, int  
Names, x  
Punctuation, COMMA  
Names, y  
Punctuation, CLOSE\_PARAN  
Punctuation, OPEN\_BRACE  
Identifier, int  
Names, X  
Punctuation, COMMA  
Names, Y  
\_\_Names, Z  
Punctuation, SEMICOLON  
Names, X  
Numerical\_Operator, EQUAL  
Real, +4.5  
Punctuation, SEMICOLON  
Names, Y  
\_\_Names, Z  
Numerical\_Operator, EQUAL  
Real, -1.2  
Punctuation, SEMICOLON  
Keyword, IF  
Punctuation, OPEN\_PARAN  
Names, x

Logical\_Operator, OR  
 Names, z  
 Punctuation, CLOSE\_PARAN  
 Names, x  
 Numerical\_Operator, EQUAL  
 Names, y  
 Punctuation, SEMICOLON  
 Keyword, ELSE  
 Names, x  
 Numerical\_Operator, EQUAL  
 Integer, 2  
 Punctuation, SEMICOLON  
 Names, x  
 Numerical\_Operator, EQUAL  
 Integer, 5  
 Punctuation, COMMA  
 Names, z  
 Numerical\_Operator, EQUAL  
 Integer, 5  
 Punctuation, SEMICOLON  
 Keyword, DO  
 Keyword, WHILE  
 Punctuation, OPEN\_PARAN  
 Names, x  
 Logical\_Operator, LESS\_THAN  
 Integer, 6  
 Punctuation, CLOSE\_PARAN  
 Punctuation, OPEN\_BRACE  
 Names, x  
 Numerical\_Operator, PLUS  
 Numerical\_Operator, PLUS  
 Punctuation, SEMICOLON  
 Names, printf  
 Punctuation, OPEN\_PARAN  
 "Names, This  
 Names, is  
 Names, a  
 Names, test  
 \Names, n  
 "Punctuation, CLOSE\_PARAN  
 Punctuation, SEMICOLON  
 Punctuation, CLOSE\_BRACE  
 Punctuation, CLOSE\_BRACE

## **2.4 Outputs**

**<Your outputs. Be sure to state which input the output is for. You can combine 2.3 and 2.4 if you wish and put the output right after the input.>**

Input and Output for the tests are shown in 2.3 above.

### **3. The Parser**

In this section, we have created a parser for our Javalet language. In this project, we have created our parser using Yacc from our Flex counterpart file using gcc as the compiler, and a Makefile for builds of the project. We have used the logic of precedence, using Yacc's %prec, to define the if-then-else statement block and to thus force a non shift-reduce (reduce-reduce) conflicts in if-then-else statements<sup>(8)</sup>. We have created our Yacc and Flex files to indent our output, in some cases, for a bit cleaner, and clearer, output by adding some verbose output to the Yacc and Lex file, without making it too cluttered. We have updated our Lex file to include the appropriate tokens needed for the grammar defined in section 3.1. We have modified the creation of the program to allow variables to be definable in the programs function definition, and we have extended the grammar to allow a user to can create multiple functions in one file in the Javalet language.

### 3.1 The Grammar

We have defined our language using the following grammar as designated from project 2. We have modified the grammar below to allow for multiple functions defined in one file of Javalet and a the addition of non-association for removal of precedence issues caused by shift-reduce in Yacc.

[ ] = 0 or 1 (regular expression: ?)

{ } = 0 or more (regular expression: \*)

< > = 1 or more (regular expression: +)

program -> method\_declaration

method\_declaration -> type name "(" ")" "{" statement\_block "}"

type -> "void" | variable\_type

variable\_type -> "int"

statement\_block -> { statement }

statement -> simple\_statement ";" | compound\_statement | "{" statement\_block "}"

simple\_statement -> declarative\_statement | assignment\_statement

declarativestatement -> variable\_type assignmentstatement

{"," assignmentstatement}

assignment\_statement -> name [assignop expression]

expression -> or\_expression

or\_expression -> and\_expression { or and\_expression }

and\_expression -> relop\_expression { and relop\_expression }

relop\_expression -> ltgt\_expression { relop ltgt\_expression }

ltgt\_expression -> addop\_expression { ltgt addop\_expression }

addop\_expression -> mulop\_expression { addop mulop\_expression }

mulop\_expression -> term { mulop term }

term -> not value | addop value | value

value -> name | number | "(" expression ")"

assignop -> "="

not -> "!"

or -> "||"

and -> "&&"

relop -> "!=" | "=="

ltgt -> ">" | "<" | ">=" | "<="

addop -> "+" | "-"

mulop -> "\*" | "/" | "%"

compound\_statement -> if\_statement | loop\_statement

if\_statement -> "if" "(" expression ")" statement [ "else" statement ]

loop\_statement -> while\_statement | dowhile\_statement | for\_statement

while\_statement -> "while" "(" expression ")" statement

do\_whilestatement -> "do" statement "while" "(" expression ")" ";"

forstatement -> "for" "(" [for\_expression] ";" [expression] ";"

[for\_expression] ")" statement

for\_expression -> declarative\_statement | assignment\_statement

{ "," assignment\_statement }

name -> letter { letter | digit | "\_" }

number -> digit { digit }

letter -> [a-zA-Z]

digit -> [0-9]

### 3.2 The Lex and Yacc files

The files below are those for the grammar defined for the Javalet language.

The **Lex file** has been extended from Project I to include power, and return tokens to the Yacc portion of this project.

```
%{
#include "y.tab.h"
%}

%%

void                return(VOID);

intlreal            return(VARTYPE);

"="                return(ASSIGNOP);
"!                 return(NOT);
"\\|"              return(OR);
"&&"               return(AND);
"!="|"=="          return(RELOP);
">"|"<"|>="|"<="  return(LTGT);

[+-]                return(ADDOP);
[*|/]              return(MULOP);
"^"                return(POWEROP);

"{"                return(OPENBRACE);
"}"                return(CLOSEBRACE);
"("                return(OPENPAREN);
")"                return(CLOSEPAREN);
","                return(COMMA);
";"                return(EOS);
if                  return(IF);
else                 return(ELSE);
while                return(WHILE);
do                  return(DO);
for                  return(FOR);
[a-zA-Z]_?([a-zA-Z0-9]_?)* { printf(" (%s) ",yytext); yylval.string=strdup(yytext);
return(NAME); }
[0-9]+               { printf(" (%s) ",yytext);yylval.integer=atoi(yytext);
return(INTEGER); }
[+-][0-9]*[.][0-9]+|+-[0-9]+[.][0-9]* { printf(" (%s) ",yytext); yylval.real=atof(yytext);
return(REAL); }
\\n\\r\\r\\n\\n\\r          /* ignore carriage returns, and line feeds */;
%%

int yywrap (void) {return 1;}
```

## The Yacc file:

We have created our Yacc file from the examples in the homework and the examples provided from the website. We have added support for multiple functions by including a program to include multiply defined functions. We have also include association rules to remove the if-then-else reduce warning created by Yacc. We have created the Yacc file in a logical manner as based off the grammar specifications for project 2. Each section correlates to the section defined by the grammar desired.

```
%{
#include <stdio.h>
%}

%start program

%union
{
    int integer;
    double real;
    char* string;
}

/* Tokens we defined in our LEXer */
%token VOID VARTYPE NAME INTEGER REAL
%token OPENPAREN CLOSEPAREN OPENBRACE CLOSEBRACE
%token ASSIGNOP NOT OR AND RELOP LTGT ADDOP MULOP POWEROP
%token IF ELSE WHILE DO FOR REDUCE
%token COMMA EOS

/* We use left and right associativity methods as provided by yacc for add, mul, power
operators*/
%left ADDOP
%left MULOP
%right POWEROP
/* Declares name for precedence token, REDUCE we use for if then else */
%nonassoc REDUCE
%nonassoc ELSE

%%

/* Program, method_dec, type, variable_type as defined from Project2 */
program: method_declarations { printf("program\n"); };
method_declarations: method_declaration | method_declarations method_declaration;
method_declaration: type NAME OPENPAREN CLOSEPAREN OPENBRACE
statement_block CLOSEBRACE { printf("method_declaration: \n"); }
                    | type NAME OPENPAREN variable_type NAME CLOSEPAREN
OPENBRACE statement_block CLOSEBRACE { printf("method_declaration: \n");};
```

```

type: VOID { printf("type: VOID\n"); } | variable_type {
printf("%s_type\n",yylval.string); };
variable_type: VARTYPE { printf("variable_type: %s\n", yylval.string); };

/* Statement_block, statemnet as defined by Project 2. Addition of empty to statement
block
as we can have an none block to start, middle or even end */
statement_block: /* none */ | statement_block statement { printf("statement_block\n"); };
statement: simple_statement EOS { printf("statement\n"); } | compound_statement {
printf("statement\n"); } | OPENBRACE statement_block CLOSEBRACE {
printf("statement\n"); };

/* Simple_statement and Declartive_statement as defined from Project2 */
simple_statement: declarative_statement { printf("simple_statement\n"); } |
assignment_statement { printf("simple_statement\n"); };
declarative_statement: variable_type assignment_statement {
printf("declarative_statement\n"); } | declarative_statement COMMA
assignment_statement { printf ("declarative_statement\n"); };
/* Assigmennt_statemnt defines a the statement as defined in Project2 specs */
assignment_statement: NAME { printf("assignment_statement: %s\n",yylval.string); } |
NAME ASSIGNOP expression { printf("assignment_statement\n"); };

/* Expression operators as defined from Project2 specs. thesei nclude or, and, ltgt, add,
mul, term, value
and the addition of power_expression which is the x^p operation */
expression: or_expression { printf("expression\n"); };
or_expression: and_expression { printf("or_expression\n"); } | or_expression OR
and_expression { printf("or_expresssion\n"); };
and_expression: relop_expression { printf("and_expression\n"); } | and_expression AND
relop_expression { printf("and_expression\n"); };
relop_expression: ltgt_expression { printf("relop_expression\n"); } | relop_expression
RELOP ltgt_expression { printf("relop_expression\n"); };
ltgt_expression: addop_expression { printf("ltgt_expression\n"); } | ltgt_expression
LTGT addop_expression { printf("ltgt_expression\n"); };
addop_expression: power_expression { printf("addop_expression\n"); } |
addop_expression ADDOP power_expression { printf("addop_expression\n"); };
mulop_expression: term { printf("mulop_expression\n"); } | mulop_expression MULOP
term { printf("mulop_expression\n"); };
term: NOT value { printf("term\n"); } | ADDOP value { printf("term\n"); } | value
{ printf("term\n"); };
value: NAME { printf("value: %s\n",yylval.string); } | INTEGER { printf("value:
%i\n",yylval.integer); } | REAL { printf("value: %d\n",yylval.real); } | OPENPAREN
expression CLOSEPAREN { printf("value: %s",yylval.string); };
/* Op not in list, but we need to add this for power to exp
Power op we have added to define to be just a regular operation or a full
powerexpression to the power */

```

```

power_expression: mulop_expression { printf("power_expression\n"); } |
power_expression POWEROP mulop_expression { printf("power_expression\n"); };

/* Compound, if, loop, while, do, for, for_exp as defined by Project2 specs*/
compound_statement: if_statement { printf("compound_statement\n"); } | loop_statement
{ printf("compound_statement\n"); };
/* here we need to add a new var REDUCT and precedence operator to make sure the if
then else operator is transposed correctly */
if_statement: IF OPENPAREN expression CLOSEPAREN statement %prec REDUCE {
printf("if_statement\n"); }
| IF OPENPAREN expression CLOSEPAREN statement ELSE statement {
printf("if_else_statement\n"); }
loop_statement: while_statement { printf("loop_statement\n"); } | dowhile_statement {
printf("dowhile_statement\n"); } | for_statement { printf("for_statement\n"); };
while_statement: WHILE OPENPAREN expression CLOSEPAREN statement {
printf("while_statement\n"); };
dowhile_statement: DO statement WHILE OPENPAREN expression CLOSEPAREN
EOS { printf("do_while_statement\n"); };
/* Here we use for statement, expression, and assignment for our: for (for_exp; exp;
for_exp) { } */
for_statement: FOR OPENPAREN for_expression EOS expression EOS for_expression
CLOSEPAREN statement { printf("for_statement\n"); };
/* Here we need to break out the for exp into an exp AND the assignment per specs of
Project 2 */
for_expression: declarative_statement { printf("for_expression\n"); } | for_assignment {
printf("for_expression\n"); };
for_assignment: assignment_statement { printf("for_assignment\n"); } | for_assignment
COMMA assignment_statement { printf("for_assignment\n"); };

%%

int main (void) {return yyparse ( );}

int yyerror (char *s) {fprintf (stderr, "%s\n", s);}

```

### 3.3 Inputs and Output

The testing files used are number input 1-4. The Input files 1-3 are those as taken from the test cases in project phase I. Test case input 4 includes test for the following (not included in tests 1-3): upper case input, OR operator input, do while input, quote's ", positive real in the input, and compound if-then-else test.

```

----- Input 1 -----
void input_a() {
    integer a, bb, xyz, b3, c, p, q;
    real b;
    a = b3;
    b = -2.5;
}

```

```

xyz = 2 + a + bb + c - p / q;
a = xyz * ( p + q );
p = a - xyz - p;
}

```

----- **Output for Input1** -----

```

type: VOID
(input_a) variable_type: input_a
(a) assignment_statement: a
declarative_statement
(bb) assignment_statement: bb
declarative_statement
(xyz) assignment_statement: xyz
declarative_statement
(b3) assignment_statement: b3
declarative_statement
(c) assignment_statement: c
declarative_statement
(p) assignment_statement: p
declarative_statement
(q) assignment_statement: q
declarative_statement
simple_statement
statement
statement_block
variable_type: q
(b) assignment_statement: b
declarative_statement
simple_statement
statement
statement_block
(a) (b3) value: b3
term
mulop_expression
power_expression
addop_expression
ltgt_expression
relop_expression
and_expression
or_expression
expression
assignment_statement
simple_statement
statement
statement_block
(b) (-2.5) value: 0

```

```

term
mulop_expression
power_expression
addop_expression
ltgt_expression
relop_expression
and_expression
or_expression
expression
assignment_statement
simple_statement
statement
statement_block
  (xyz) (2) value: 2
term
mulop_expression
power_expression
addop_expression
  (a) value: a
term
mulop_expression
power_expression
addop_expression
  (bb) value: bb
term
mulop_expression
power_expression
addop_expression
  (c) value: c
term
mulop_expression
power_expression
addop_expression
  (p) value: p
term
mulop_expression
  (q) value: q
term
mulop_expression
power_expression
addop_expression
ltgt_expression
relop_expression
and_expression
or_expression
expression

```

```

assignment_statement
simple_statement
statement
statement_block
    (a) (xyz) value: xyz
term
mulop_expression
    (p) value: p
term
mulop_expression
power_expression
addop_expression
    (q) value: q
term
mulop_expression
power_expression
addop_expression
ltgt_expression
relop_expression
and_expression
or_expression
expression
value: qterm
mulop_expression
power_expression
addop_expression
ltgt_expression
relop_expression
and_expression
or_expression
expression
assignment_statement
simple_statement
statement
statement_block
    (p) (a) value: a
term
mulop_expression
power_expression
addop_expression
    (xyz) value: xyz
term
mulop_expression
power_expression
addop_expression
    (p) value: p

```

term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression  
and\_expression  
or\_expression  
expression  
assignment\_statement  
simple\_statement  
statement  
statement\_block  
method\_declaration:  
program

----- **Input 2** -----

```
void input_b() {  
  if ( i > j )  
    i = i + j;  
  else if ( i < j )  
    i = 1;  
}
```

----- **Output for input 2** -----

```
type: VOID  
  (input_b)    (i) value: i  
term  
mulop_expression  
  power_expression  
addop_expression  
ltgt_expression  
  (j) value: j  
term  
mulop_expression  
  power_expression  
addop_expression  
ltgt_expression  
relop_expression  
and_expression  
or_expression  
expression  
  (i) (i) value: i  
term  
mulop_expression  
  power_expression  
addop_expression  
  (j) value: j  
term  
mulop_expression  
power_expression  
addop_expression  
ltgt_expression  
relop_expression  
and_expression  
or_expression  
xpression  
assignment_statement  
simple_statement  
statement  
  (i) value: i
```

term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
(j) value: j  
term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression  
and\_expression  
or\_expression  
expression  
(i) (1) value: 1  
term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression  
and\_expression  
or\_expression  
expression  
assignment\_statement  
simple\_statement  
statement  
if\_statement  
compound\_statement  
statement  
if\_else\_statement  
compound\_statement  
statement  
statement\_block  
method\_declaration:  
program

-----**Input 3**-----

```
void input_c() {  
    while ( i < j && j < k ) {  
        k = k + 1;  
        while ( i == j )  
            i = i + 2;  
    }  
}
```

----- **Output for Input 3** -----

```
type: VOID  
  (input_c)    (i) value: i  
term  
mulop_expression  
  power_expression  
addop_expression  
ltgt_expression  
  (j) value: j  
term  
mulop_expression  
  power_expression  
addop_expression  
ltgt_expression  
relop_expression  
and_expression  
  (j) value: j  
term  
mulop_expression  
  power_expression  
addop_expression  
ltgt_expression  
  (k) value: k  
term  
mulop_expression  
  power_expression  
addop_expression  
ltgt_expression  
relop_expression  
and_expression  
or_expression  
expression  
  (k) (k) value: k  
term  
mulop_expression  
  power_expression  
addop_expression
```

(1) value: 1  
term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression  
and\_expression  
or\_expression  
expression  
assignment\_statement  
simple\_statement  
statement  
statement\_block

(i) value: i  
term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression

(j) value: j  
term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression  
and\_expression  
or\_expression  
expression

(i) (i) value: i  
term  
mulop\_expression  
power\_expression  
addop\_expression

(2) value: 2  
term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression  
and\_expression  
or\_expression  
expression

assignment\_statement  
simple\_statement  
statement  
while\_statement  
loop\_statement  
compound\_statement  
statement  
statement\_block  
    statement  
while\_statement  
loop\_statement  
compound\_statement  
statement  
statement\_block  
method\_declaration:  
program

----- **Input 4: An Example of your own** -----  
**Testing compound if then else statement along with multiple functions:**

```
void test(int u)
{
  int X;
  int Y;

  X=1; Y=-1;
  if (X || Y<0)
  {
    for (int x=1;x<255;Y=Y^X+Y)
    {
      if (Y%2==0)
      {
        do Y=Y-1;
        while(Y%2!=0);
      }
    }
  }

  if (!x_1)
  {
    x_1=1;
  }
  else
  if (x_3)
    x_2=1;
}
```

----- **Output for Input 4** -----

```
type: VOID
(test) variable_type: test
(u) variable_type: u
(X) assignment_statement: X
declarative_statement
simple_statement
statement
statement_block
  variable_type: X
  (Y) assignment_statement: Y
declarative_statement
simple_statement
statement
statement_block
```

(X) (1) value: 1  
 term  
 mulop\_expression  
 power\_expression  
 addop\_expression  
 ltgt\_expression  
 relop\_expression  
 and\_expression  
 or\_expression  
 expression  
 assignment\_statement  
 simple\_statement  
 statement  
 statement\_block  
 (Y) (1) value: 1  
 term  
 mulop\_expression  
 power\_expression  
 addop\_expression  
 ltgt\_expression  
 relop\_expression  
 and\_expression  
 or\_expression  
 expression  
 assignment\_statement  
 simple\_statement  
 statement  
 statement\_block  
 (X) value: X  
 term  
 mulop\_expression  
 power\_expression  
 addop\_expression  
 ltgt\_expression  
 relop\_expression  
 and\_expression  
 or\_expression  
 (Y) value: Y  
 term  
 mulop\_expression  
 power\_expression  
 addop\_expression  
 ltgt\_expression  
 (0) value: 0  
 term  
 mulop\_expression

power\_expression  
 addop\_expression  
 ltgt\_expression  
 relop\_expression  
 and\_expression  
 or\_expresssion  
 expression  
     variable\_type: (null)  
 (x) (1) value: 1  
 term  
 mulop\_expression  
 power\_expression  
 addop\_expression  
 ltgt\_expression  
 relop\_expression  
 and\_expression  
 or\_expression  
 expression  
 assignment\_statement  
 declarative\_statement  
 for\_expression  
 (x) value: x  
 term  
 mulop\_expression  
 power\_expression  
 addop\_expression  
 ltgt\_expression  
 (255) value: 255  
 term  
 mulop\_expression  
 power\_expression  
 addop\_expression  
 ltgt\_expression  
 relop\_expression  
 and\_expression  
 or\_expression  
 expression  
 (Y) (Y) value: Y  
 term  
 mulop\_expression  
 power\_expression  
 (X) value: X  
 term  
 mulop\_expression  
 power\_expression  
 addop\_expression

(Y) value: Y  
term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression  
and\_expression  
or\_expression  
expression  
assignment\_statement  
for\_assignment  
for\_expression

(Y) value: Y

term  
mulop\_expression  
(2) value: 2  
term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression  
(0) value: 0  
term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression  
and\_expression  
or\_expression  
expression

(Y) (Y) value: Y

term  
mulop\_expression  
power\_expression  
addop\_expression  
(1) value: 1  
term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression  
and\_expression

or\_expression  
 expression  
 assignment\_statement  
 simple\_statement  
 statement  
     (Y) value: Y  
 term  
 mulop\_expression  
     (2) value: 2  
 term  
 mulop\_expression  
 power\_expression  
 addop\_expression  
 ltgt\_expression  
 relop\_expression  
     (0) value: 0  
 term  
 mulop\_expression  
 power\_expression  
 addop\_expression  
 ltgt\_expression  
 relop\_expression  
 and\_expression  
 or\_expression  
 expression  
 do\_while\_statement  
 dowhile\_statement  
 compound\_statement  
 statement  
 statement\_block  
     statement  
         if\_statement  
 compound\_statement  
 statement  
 statement\_block  
 statement  
 for\_statement  
 for\_statement  
 compound\_statement  
 statement  
 statement\_block  
     statement  
         if\_statement  
 compound\_statement  
 statement  
 statement\_block

(x\_1) value: x\_1  
term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression  
and\_expression  
or\_expression  
expression  
                  (x\_1) (1) value: 1

term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression  
and\_expression  
or\_expression  
expression  
assignment\_statement  
simple\_statement  
statement  
statement\_block  
    statement  
        (x\_3) value: x\_3

term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression  
and\_expression  
or\_expression  
expression  
                  (x\_2) (1) value: 1

term  
mulop\_expression  
power\_expression  
addop\_expression  
ltgt\_expression  
relop\_expression  
and\_expression  
or\_expression  
expression  
assignment\_statement

simple\_statement  
statement  
if\_statement  
compound\_statement  
statement  
if\_else\_statement  
compound\_statement  
statement  
statement\_block  
method\_declaration:  
program

#### **4. Abstract Syntax Tree**

In this section we define the attributes created to define an Abstract Syntax Tree (AST.) Using the parsed tokens we create an AST and simply output a logical view of a programs input. This AST provides the logical breakdown using Lex and Yacc's internal functional features to create a simple binary AST data structure. We have used the structure, as taken from the class examples, to represent the data points for a Javalet program, to be that of operators and operands down the AST. We have included the updated Lex code which was required to create this AST as well as the updated Yacc code to create the structure and AST. We have run the three example inputs and the output is shown in section 4.3 along with the input data.

#### 4.1 AST Lex file update.

The AST Lex file, shown below, has additional attributes required to create the AST. We have included the node data structure, as taken from class, and included a `line_number` counter and `line_number` local variable for counting lines (logic and idea as taken from class resources.) We have updated the logic of the Lex file to include the allocation of 'node' space and the assignments of the parsed Lex type, as well as the update of the line number where the action has taken place. We can then take these actions and pass this data to our Yacc files (section 4.2) for the AST processing and output.

##### Lex file

```
%{
#include "y.tab.h"
#include <string.h>

typedef struct node
{
    struct node *left;
    struct node *right;
    char *token;
    int line_number;
} node;

extern struct node *yylval;

int line_number = 1;

%}
%%
void          { yylval = malloc(sizeof(node));
               ((node*)yylval)->token=strdup(yytext);
               ((node*)yylval)->left=0;
               ((node*)yylval)->right=0;
               ((node*)yylval)->line_number=line_number;
               return VOID; }
intlreal     { yylval = malloc(sizeof(node));
               ((node*)yylval)->token=strdup(yytext);
               ((node*)yylval)->left=0;
               ((node*)yylval)->right=0;
               ((node*)yylval)->line_number=line_number;
               return VARTYPE; }
if          { yylval = malloc(sizeof(node));
               ((node*)yylval)->token=strdup(yytext);
               ((node*)yylval)->left=0;
               ((node*)yylval)->right=0;
               ((node*)yylval)->line_number=line_number;
               return IF; }
```

```

else      { yylval = malloc(sizeof(node));
          ((node*)yylval)->token=strdup(yytext);
          ((node*)yylval)->left=0;
          ((node*)yylval)->right=0;
          ((node*)yylval)->line_number=line_number;
          return ELSE;}

while    { yylval = malloc(sizeof(node));
          ((node*)yylval)->token=strdup(yytext);
          ((node*)yylval)->left=0;
          ((node*)yylval)->right=0;
          ((node*)yylval)->line_number=line_number;
          return WHILE;}

do       { yylval = malloc(sizeof(node));
          ((node*)yylval)->token=strdup(yytext);
          ((node*)yylval)->left=0;
          ((node*)yylval)->right=0;
          ((node*)yylval)->line_number=line_number;
          return DO;}

for      { yylval = malloc(sizeof(node));
          ((node*)yylval)->token=strdup(yytext);
          ((node*)yylval)->left=0;
          ((node*)yylval)->right=0;
          ((node*)yylval)->line_number=line_number;
          return FOR;}

"{"     return(OPENBRACE);
"}"     return(CLOSEBRACE);
"("     return(OPENPAREN);
")"     return(CLOSEPAREN);
","     return(COMMA);
";"     return(EOS);
[a-zA-Z]_?([a-zA-Z0-9]+_?)*  { yylval = malloc(sizeof(node));
                             ((node*)yylval)->token=strdup(yytext);
                             ((node*)yylval)->left=0;
                             ((node*)yylval)->right=0;
                             ((node*)yylval)->line_number=line_number;
                             return NAME;}

[0-9]+  { yylval = malloc(sizeof(node));
          ((node*)yylval)->token=strdup(yytext);
          ((node*)yylval)->left=0;
          ((node*)yylval)->right=0;
          ((node*)yylval)->line_number=line_number;
          return INTEGER;}

[+][0-9]*[.][0-9]+  { yylval = malloc(sizeof(node));
                     ((node*)yylval)->token=strdup(yytext);
                     ((node*)yylval)->left=0;
                     ((node*)yylval)->right=0;

```

```

((node*)yyval)->line_number=line_number;
return REAL;}
[+][0-9]+[.][0-9]*  {yyval = malloc(sizeof(node));
((node*)yyval)->token=strdup(yytext);
((node*)yyval)->left=0;
((node*)yyval)->right=0;
((node*)yyval)->line_number=line_number;
return REAL;}
"="
{yyval = malloc(sizeof(node));
((node*)yyval)->token=strdup(yytext);
((node*)yyval)->left=0;
((node*)yyval)->right=0;
((node*)yyval)->line_number=line_number;
return ASSIGNOP;}
"!"
{yyval = malloc(sizeof(node));
((node*)yyval)->token=strdup(yytext);
((node*)yyval)->left=0;
((node*)yyval)->right=0;
((node*)yyval)->line_number=line_number;
return NOT;}
"\\|"
{yyval = malloc(sizeof(node));
((node*)yyval)->token=strdup(yytext);
((node*)yyval)->left=0;
((node*)yyval)->right=0;
((node*)yyval)->line_number=line_number;
return OR;}
"&&"
{yyval = malloc(sizeof(node));
((node*)yyval)->token=strdup(yytext);
((node*)yyval)->left=0;
((node*)yyval)->right=0;
((node*)yyval)->line_number=line_number;
return AND;}
"!="|"=="
{yyval = malloc(sizeof(node));
((node*)yyval)->token=strdup(yytext);
((node*)yyval)->left=0;
((node*)yyval)->right=0;
((node*)yyval)->line_number=line_number;
return RELOP;}
">"|"<"|>="|"<="  {yyval = malloc(sizeof(node));
((node*)yyval)->token=strdup(yytext);
((node*)yyval)->left=0;
((node*)yyval)->right=0;
((node*)yyval)->line_number=line_number;
return LTGT;}
[+-]
{yyval = malloc(sizeof(node));
((node*)yyval)->token=strdup(yytext);

```

```

((node*)yyval)->left=0;
((node*)yyval)->right=0;
((node*)yyval)->line_number=line_number;
return ADDOP;}
[*/%] {yyval = malloc(sizeof(node));
((node*)yyval)->token=strdup(yytext);
((node*)yyval)->left=0;
((node*)yyval)->right=0;
((node*)yyval)->line_number=line_number;
return MULOP;}
"^" {yyval = malloc(sizeof(node));
((node*)yyval)->token=strdup(yytext);
((node*)yyval)->left=0;
((node*)yyval)->right=0;
((node*)yyval)->line_number=line_number;
return POWEROP;}
[ \t]+ /* ignore whitespace */;
"\n" line_number++;
%%

int yywrap (void) {return 1;}

```

## 4.2 AST Yacc file update.

The AST Yacc file, shown below, has additional attributes required to create the AST and output an AST from input. We have included the node data structure, as taken from class, and included a `line_number` counter and `line_number` local variable for counting lines (logic from class resources.) We have updated the logic of the Yacc file to include the allocation of 'node' space and the assignments from the Lex file to create the AST structure. The Yacc file creates the AST by logically updating and adding the tree items to the node list. We use the Lex line number(s), see above, to indicate where an action has taken place. We use a modified print tree routine to output the AST in a clean line based manner. The allocation using the node structure and `mknode` function assists us to create the AST tree structure. The creation of the AST is pretty straight forward for most of the operations define in the Yacc file. The tree can be simply created by adding the type of operator with the operation on that operator, or in some cases just the operation itself. The logical statements `if` and `for` require additional allocation and logic. This is required to allow for the logic portion of the statement. An example of this, is an `if-then-else` conditional where we have to deal with not only the `if` portion but also the `else` and statement portions, so in this case, as is similar in the looping for case, we need to allocate memory for a branch as we are adding node based logic to the tree. See the below Yacc file for more details.

Yacc file

```
%{
#include <stdio.h>
#include <string.h>

// Data structure used to hold our tree struct
// we hold the branches left and right, and the item as a "char"
// line_number is our logic to print which line we saw a token(s)
typedef struct node
{
    struct node *left;
    struct node *right;
    char *token;
    int line_number;
} node;

// we need to tell Yacc what our dstruct is by setting YYSTYPE to our
// node structure
#define YYSTYPE struct node *

// local function defs
node *mknode(node *left, node *right, char *token);
void printtree(node *tree, int d);
void printtree2(node *tree, int d);

// We count the line numbers starting at 1
int yacc_line_number = 1;

%}
```

```

%start program

/* Tokens we defined in our LEXer */
%token VOID VARTYPE NAME INTEGER REAL
%token OPENPAREN CLOSEPAREN OPENBRACE CLOSEBRACE
%token ASSIGNOP NOT OR AND POWEROP RELOP LTGT ADDOP MULOP
%token IF REDUCEIF ELSE WHILE DO FOR
%token COMMA EOS

/* We use left and right associativity methods as provided by yacc for add,mul,power
operators*/
%left ADDOP
%left MULOP
%right POWEROP

/* Declares name for precedence token,REDUCE we use for if then else */
%nonassoc REDUCEIF
%nonassoc ELSE

%%

/* Program,method_dec,type,variable_type as defined from Project2 */
// we allow for multiple procedures/methods
program:      method_declarations { printtree($1,0); };
method_declarations: method_declaration { $$=$1; } | method_declarations
method_declaration { $$=mknode($1,$2,""); };
method_declaration: type NAME OPENPAREN CLOSEPAREN OPENBRACE
statement_block CLOSEBRACE { $1->left=$2; $1->right=$6; $$=$1; }
| type NAME OPENPAREN variable_type NAME CLOSEPAREN
OPENBRACE statement_block CLOSEBRACE { $1->left=$2; $1->right=$8; $$=$1; };
type:      VOID { $$=$1; } | variable_type { $$=$1; };
variable_type:  VARTYPE { $$=$1; };

/* Statement_block,statement as defined by Project 2. Addition of empty to statement
block
as we can have an none block to start,middle or even end */
statement_block: /* none */ { $$=0; } | statement_block statement {
$$=mknode($1,$2,""); };
statement: simple_statement EOS { $$=$1; } | compound_statement { $$=$1; } |
OPENBRACE statement_block CLOSEBRACE { $$=$2; };

/* Simple_statement and Declarative_statement as defined from Project2 */
simple_statement: declarative_statement { $$=$1; } | assignment_statement { $$=$1; };
declarative_statement: variable_type assignment_statement { $1->left=$2; $$=$1; } |
declarative_statement COMMA assignment_statement { $$=mknode($1,$3,""); };
/* Assignment_statement defines a the statement as defined in Project2 specs */

```

```
assignment_statement: NAME { $$=$1; } | NAME ASSIGNOP expression { $2->left=$1;
$2->right=$3; $$=$2;};
```

```
/* Expression operators as defined from Project2 specs. these include
or,and,ltgt,add,mul,term,value
```

```
and the addition of power_expression which is the x^p operation */
```

```
expression: or_expression { $$=$1; };
or_expression: and_expression { $$=$1; } | or_expression OR and_expression {
$2->left=$1; $2->right=$3; $$=$2; };
```

```
and_expression: relop_expression { $$=$1; } | and_expression AND relop_expression
{ $2->left=$1; $2->right=$3; $$=$2; };
```

```
relop_expression: ltgt_expression { $$=$1; } | relop_expression RELOP ltgt_expression
{ $2->left=$1; $2->right=$3; $$=$2; };
```

```
ltgt_expression: addop_expression { $$=$1; } | ltgt_expression LTGT addop_expression
{ $2->left=$1; $2->right=$3; $$=$2; };
```

```
addop_expression: power_expression { $$=$1; } | addop_expression ADDOP
power_expression { $2->left=$1; $2->right=$3; $$=$2; };
```

```
/* Op not in list,but we need to add this for power to exp
```

```
Power op we have added to define to be just a regular operation or a full
powerexpression to the power */
```

```
power_expression: mulop_expression { $$=$1; } | power_expression POWEROP
mulop_expression { $2->left=$1; $2->right=$3; $$=$2; };
```

```
mulop_expression: term { $$=$1; } | mulop_expression MULOP term {
$2->left=$1; $2->right=$3; $$=$2; };
```

```
term: NOT value { $1->left=$2; $$=$1; } | ADDOP value { $1->left=$2; $$=$1; } |
value { $$=$1; };
```

```
value: NAME { $$=$1; } | INTEGER { $$=$1; } | REAL { $$=$1; } |
OPENPAREN expression CLOSEPAREN { $$=$2; };
```

```
/* Compound,if,loop,while,do,for,for_exp as defined by Project2 specs*/
```

```
compound_statement: if_statement { $$=$1; } | loop_statement { $$=$1; };
```

```
/* here we need to add a new var REDUCT and precedence operator to make sure the if
then else operator is transposed correctly */
```

```
if_statement: IF OPENPAREN expression CLOSEPAREN statement %prec
REDUCEIF { $1->left=$3; $1->right=$5; $$=$1; }
```

```
| IF OPENPAREN expression CLOSEPAREN statement ELSE statement {
$1->left=$3; $1->right=mknnode($5,$6,""); $6->left=$7; $$=$1; };
```

```
loop_statement: while_statement { $$=$1; } | dowhile_statement { $$=$1; }
```

```
for_statement { $$=$1; };
```

```
while_statement: WHILE OPENPAREN expression CLOSEPAREN statement
{ $1->left=$3; $1->right=$5; $$=$1; };
```

```
dowhile_statement: DO statement WHILE OPENPAREN expression CLOSEPAREN
EOS { $1->left=$2; $1->right=$3; $3->left=$5; $$=$1; };
```

```
/* Here we use for statement,expression,and assignment for our: for (for_exp; exp;
for_exp) { } */
```

```

/* This gets a bit more complicated to make the node for left/right as we need to create
the for statement and add the expression
Thus the need for a nested allocation on the left side of the for_expression by adding in
the expression to the list of the nodes */
for_statement:  FOR OPENPAREN for_expression EOS expression EOS
for_expression CLOSEPAREN statement { $1->left=mknnode($3,mknnode($5,$7,""), "");
$1->right=$9; $$=$1; };
/* Here we need to break out the for exp into an exp AND the assignment per specs of
Project 2 */
for_expression:  declarative_statement { $$=$1; }| for_assignment { $$=$1; };
for_assignment:  assignment_statement { $$=$1; }| for_assignment COMMA
assignment_statement { $$=mknnode($1,$3,""); };

%%

int main (void) {return yyparse ( );}

/* Function: mknnode
Inputs: left,right,token
left,right - binary tree branches
token - item taken from lexer
Credits: As taken and added to from lab3 sources
*/
node *mknnode(node *left,node *right,char *token)
{
/* malloc the node */
node *newnode = (node *)malloc(sizeof(node));
char *newstr = (char *)malloc(strlen(token)+1);
strcpy(newstr,token);
newnode->left = left;
newnode->right = right;
newnode->token = newstr;
return(newnode);
}

/* Function: printtree
Inputs: node,depth
node - our binary data structure node
depth - how deep have we already gone (used for line #'s)
Credits: As taken from lab3 sources and course docs provided
*/
void printtree(node *tree,int depth)
{
int i;
static LINE1=0;

```

```

// If the tree is null return
if(!tree)
{
    return;
}

// are we at a point where should print a line?
// if we have gone past the local line,then print it once
if (tree->line_number > yacc_line_number)
{
    printf("\nLine(%d)",tree->line_number);
    yacc_line_number = tree->line_number;
}

// logic for print line1
if (tree->left || tree->right)
{
    if (tree->line_number==1 && !LINE1)
    {
        printf("\nLine(%d)",tree->line_number);
        LINE1=1;
    }
    printf("\n");
    for(i = 0; i <= depth; i++)
    {
        printf("_");
    }
    printf("(");
}

// print the actual item
printf(" %s ",tree->token);

// depth left
if (tree->left)
{
    printtree(tree->left,depth+1);
}

// depth right
if (tree->right)
{
    printtree(tree->right,depth+1);
}

// if both braches then we can output a )

```

```
    if (tree->left || tree->right)
    {
        printf("");
    }
}

int yyerror (char *s) {fprintf (stderr,"%s\n",s);}
```

### **4.3 Input and Output**

The following section has the inputs 1-3 provided and an input 4 for testing. We output an AST from these inputs. We have separated the input and output into one listing broken up by page breaks.



```
_____( * xyz
_____( + p q )))
Line(8)
_____( = p
_____( -
_____( - a xyz ) p )))
```

**Input 2:**

As provided from project 1

```

void input_b() {
  if ( i > j )
    i = i + j;
  else if ( i < j )
    i = 1;
}

```

**OUTPUT Input 2:**

Line(1)

\_( void input\_b

\_\_

Line(2)

\_\_( if

\_\_\_\_(&gt; i j )

\_\_\_\_

Line(3)

\_\_\_\_( = i

\_\_\_\_(+ i j ))

Line(4)

\_\_\_\_( else

\_\_\_\_( if

\_\_\_\_(&lt; i j )

Line(5)

\_\_\_\_( = i 1 ))))))))

**Input 3:**

As provided from project 1

```
void input_c() {
  while ( i < j && j < k ) {
    k = k + 1;
    while ( i == j )
      i = i + 2;
  }
}
```

**OUTPUT Input 3:**

Line(1)

\_( void input\_c

\_\_(  
Line(2)

\_\_( while

\_\_\_\_( &&

\_\_\_\_(< i j )

\_\_\_\_(< j k ))

\_\_\_\_(  
\_\_\_\_(  
Line(3)

\_\_\_\_( = k

\_\_\_\_(+ k 1 )))

Line(4)

\_\_\_\_( while

\_\_\_\_( == i j )

Line(5)

\_\_\_\_( = i

\_\_\_\_(+ i 2 ))))))))

#### Input 4:

Our own Testing input

```
void test(int varg)
{
    int X;
    int Y;

    X=1;
    Y=-1;
    if (X || Y<0)
    {
        for (int x=1;x<255;Y=Y^X+Y)
        {
            if (Y%2==0)
            {
                do Y=Y-1;
                while(Y%2!=0);
            }
        }
    }

    if (!x_1)
    {
        x_1=1;
    }
    else
    if (x_3)
        x_2=1;
}
}
```

#### OUTPUT Input 4

```
Line(1)
_( void test
__(
__(
__(
__(
__(
Line(3)
_____( int X ))
Line(4)
_____( int Y ))
Line(6)
_____( = X 1 ))
```

```

Line(7)
_____( = Y
_____( - 1 )))
Line(8)
_____( if
_____( || X
_____( < Y 0 ))
_____(
Line(10)
_____( for
_____(
_____( int
_____( = x 1 ))
_____(
_____( < x 255 )
_____( = Y
_____( +
_____( ^ Y X ) Y ))))
_____(
Line(12)
_____( if
_____( ==
_____( % Y 2 ) 0 )
_____(
Line(14)
_____( do
_____( = Y
_____( - Y 1 ))
Line(15)
_____( while
_____( !=
_____( % Y 2 ) 0 ))))))))
Line(20)
_____( if
_____( ! x_1 )
_____(
_____(
Line(22)
_____( = x_1 1 ))
Line(24)
_____( else
Line(25)
_____( if x_3
Line(26)
_____( = x_2 1 ))))))))

```

## Bibliography

1. **Microsystems, Sun.** Java. *The Java History Timeline*. [Online] Sun Microsystems, 1991. <http://www.java.com/en/javahistory/timeline.jsp>.
2. **t, Vern Paxson.** flex: The Fast Lexical Analyzer. *FLEX*. [Online] GNU, 1987.
3. **Schmidt, M. E. Lesk and E.** LEX – Lexical Analyzer Generator. *LEX*. [Online] <http://opengroup.org/onlinepubs/007908775/xcu/lex.html>.
4. **Stephen C. Johnson.** YACC. *Yet Another Compiler Compiler*. [Online] AT&T, 1988. <http://dinosaur.compilertools.net/yacc/index.html>.
5. **Stallman, Richard M.** GNU GCC. *GNU GCC*. [Online] 1987. <http://gcc.gnu.org/wiki/History>.
6. **Stallman, Richard.** The GNU Project. [Online] 1984. <http://www.gnu.org/gnu/thegnuproject.html>.
7. **License, GNU Free Documentation.** GNU Make. *GNU*. [Online] 1988. <http://www.gnu.org/software/make/manual/make.html#Top>.
8. **Parsing with Yacc. Precedence.** [Online] SCO, 2004. [http://ou800doc.caldera.com/en/SDK\\_tools/\\_Precedence.html](http://ou800doc.caldera.com/en/SDK_tools/_Precedence.html).