

Compilers summer 2009
Project Part 2
Alan Ritacco

Contents

| | |
|---|----|
| 1. Introduction | 2 |
| 2. The Scanner | 3 |
| 2.1 The Tokens | 3 |
| 2.2 The Lex File | 4 |
| 2.3 Inputs | 5 |
| 2.4 Outputs | 11 |
| 3. The Parser | 12 |
| 3.1 The Grammar | 13 |
| 3.2 The Lex and Yacc files | 15 |
| 3.3 Inputs and Output | 16 |
| Bibliography | 35 |

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⁽¹⁾ programming language. In project phase I we have created a scanner written using the fast lexical analyzer (Flex⁽²⁾) parlance, which is based off of Lex⁽³⁾. 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⁽⁴⁾. 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⁽⁵⁾ version 4.3 for the lexicographical and compilers respectively. A GNU⁽⁶⁾ Makefile⁽⁷⁾ 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.

2.2 The Lex File

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

DIGIT  [0-9]
LETTER [a-zA-Z]

%%

void      printf("Identifier, %s\n",yytext);
int       printf("Identifier, %s\n",yytext);
real     printf("Identifier, %s\n",yytext);
\!       printf("Logical_Operator, NOT\n");
\\       printf("Logical_Operator, OR\n");
\&\&    printf("Logical_Operator, AND\n");
==       printf("Logical_Operator, EQUAL\n");
\!=     printf("Logical_Operator, NOT_EQUAL\n");
\<      printf("Logical_Operator, LESS_THAN\n");
\<=     printf("Logical_Operator, LESS_OR_EQUAL\n");
>       printf("Logical_Operator, GREATER_THAN\n");
>=     printf("Logical_Operator, GREATER_OR_EQUAL\n");
\+      printf("Numerical_Operator, PLUS\n");
\|-     printf("Numerical_Operator, MINUS\n");
\*      printf("Numerical_Operator, MULTIPLY\n");
\/      printf("Numerical_Operator, DIVIDE\n");
\=     printf("Numerical_Operator, EQUAL\n");
\{      printf("Punctuation, OPEN_BRACE\n");
\}      printf("Punctuation, CLOSE_BRACE\n");
\(\     printf("Punctuation, OPEN_PARAN\n");
\)      printf("Punctuation, CLOSE_PARAN\n");
\,      printf("Punctuation, COMMA\n");
;       printf("Punctuation, SEMICOLON\n");
if      printf("Keyword, IF\n");
else    printf("Keyword, ELSE\n");
while   printf("Keyword, WHILE\n");
do      printf("Keyword, DO\n");
for     printf("Keyword, FOR\n");

{LETTER}(\_?(((LETTER)|{DIGIT}))+)*      printf("Names, %s\n",yytext);
{DIGIT}+      printf("Integer, %s\n",yytext);
[+-]{DIGIT}+.{DIGIT}+      printf("Real, %s\n",yytext);
[ \t]+        /* ignore whitespace */;
\n\r\n\r\n\r  /* ignore carriage returns, and line feeds */;

%%

int main (void) {yylex(); return 0;}
int yywrap (void) {return 1;}
```

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⁽⁸⁾. 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"); };
/* Assignemnt_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

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.