**Compiler Project Report**

For this project, our group has developed a programming language that is similar to C++. We called it Pi++. It is much smaller than C++ in the sense that we took several parts of C++ and used only those parts for our language. In essence our language is somewhat similar to small C, but we do have less constructs in our language and we used different names for some of our constructs.

**Grammar File**

grammar Pi++;

@lexer::members {

int nesting = 0;

}

@header {

#include "wci/intermediate/TypeSpec.h"

using namespace wci::intermediate;

}

prog: stat+ ;

stat: assign

| expr

| declaration

| if\_stm

| while\_stm

| print

| pass\_by\_value

| NEWLINE

;

if\_stm: IF '(' expr ')' stat ;

while\_stm: WHILE '(' expr ')' stat ;

print: PRINT INSERTION out;

out: variable\_id | STRING;

declaration: decl | func;

return\_type: ID;

returning: 'return' ID;

func: return\_type ':' ID '(' assign ')' '{' stat+ '}' returning ;

decl: type\_id variable\_list ;

variable\_list: variable\_id ;

variable\_id: ID;

type\_id: ID;

assign: variable\_id '=' expr | declaration '=' expr ;

expr locals [ TypeSpec \*type = nullptr ]

: expr MUL\_DIV\_OP expr # mul\_div\_op

| expr ADD\_SUB\_OP expr # add\_sub\_op

| INT # integer

| CHAR # char

| ID # identifier

| '(' expr ')' #parentheses

| expr EQUALITY expr #equality

;

EQUALITY: '===' ;

MUL\_DIV\_OP: MUL\_OP | DIV\_OP ;

ADD\_SUB\_OP: ADD\_OP | SUB\_OP ;

MUL\_OP: '\*' ;

DIV\_OP: '/' ;

ADD\_OP: '+' ;

SUB\_OP: '-' ;

INT: [0-9]+;

CHAR: '\''[ a-zA-Z0-9\p{P}]'\'';

IF: 'if' ;

WHILE: 'while' ;

PRINT: 'print' ;

INSERTION: '<<' ;

STRING: '"'[ a-zA-Z0-9\p{P}][ a-zA-Z0-9\p{P}]\*'"';

ID: [a-zA-Z\_][a-zA-Z\_0-9]\* ;

COMMENT: '#' ~[\r\n]\* ->skip ;

WS: [ \t] -> skip ;

NEWLINE: '\r'? '\n' {nesting==0}?;

LINE\_ESCAPE: '\\' '\r'? '\n' -> skip ;

IGNORE\_NEWLINE: '\r'? '\n' {nesting>0}? -> skip ;

LPAREN: '(' {nesting++;} ;

RPAREN: ')' {nesting--;} ;

LBRACK: '[' {nesting++;} ;

RBRACK: ']' {nesting--;} ;

**Description**

Our programming language is similar to C++, but it is also much smaller than C++. We changed some of the syntax as well as the names of some constructs. We managed to achieve this by using antlr syntax to generate the desired functionality. The programming language can do sophisticated adding , subtraction , multiplication , loop statement and functions. We can call functions and pass parameters by value as well.

**Code Templates**

IF Statement:

Our Code:

if(x===5) x=1

Jasmin Code:

getstatic sample/x I

istore 1

iload 1

ldc 5

istore 2

iload 2

if\_icmpeq L001

goto L002

L001:

ldc 1

putstatic sample/x I

L002:

WHILE Statement:

Our code:

while(x===1) if(x===1) x=7

Jasmin Code:

getstatic sample/x I

istore 3

ldc 1

istore 4

WHILE:

iload 3

iload 4

if\_icmpne END\_WHILE

goto EQUAL

EQUAL:

getstatic sample/x I

istore 1

iload 1

ldc 1

istore 2

iload 2

if\_icmpeq L003

goto L004

L003:

ldc 7

putstatic sample/x I

L004:

END\_WHILE:

FUNCTION with Call (pass\_by\_value) and Return:Our Code:

int : func (int x=0) {x=3

int y=2

} return x

func(x)

Jasmin Code:

Code for function:

.method public static func(I)I

iload 0

istore 1

ldc 3

istore 1

ldc 2

istore 2

iload 1

ireturn

.limit stack 16

.limit locals 16

.end method

Code for function call:

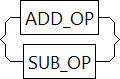
invokestatic sample/func(I)I

istore 2

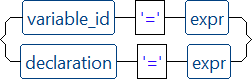
iload 2

putstatic sample/x I

**Syntax Diagrams**



ADD\_SUP\_OP



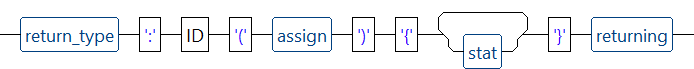
ASSIGN



CALL



EQUALITY



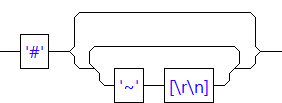
FUNC



INSERTION



CHAR



COMMENT



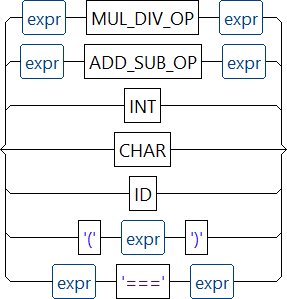
DECL



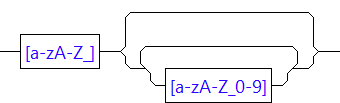
DECLARATION



DIV\_OP



EXPR



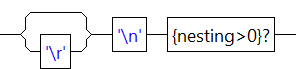
ID



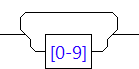
IF



IF\_STMT



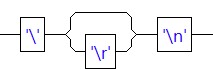
IGNORE\_NEWLINE



INT

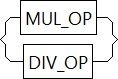


LBRACK





LPAREN

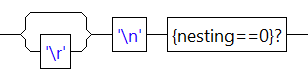




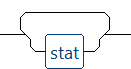
MUL\_OP



ADD\_OP



NEWLINE



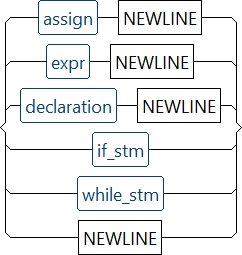
PROG



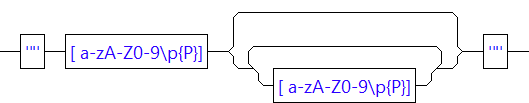
RBRACK



RPAREN



STAT



STRING



SUB\_OP



TYPE\_ID



VARIABLE\_ID



WHILE



WHILE\_STM



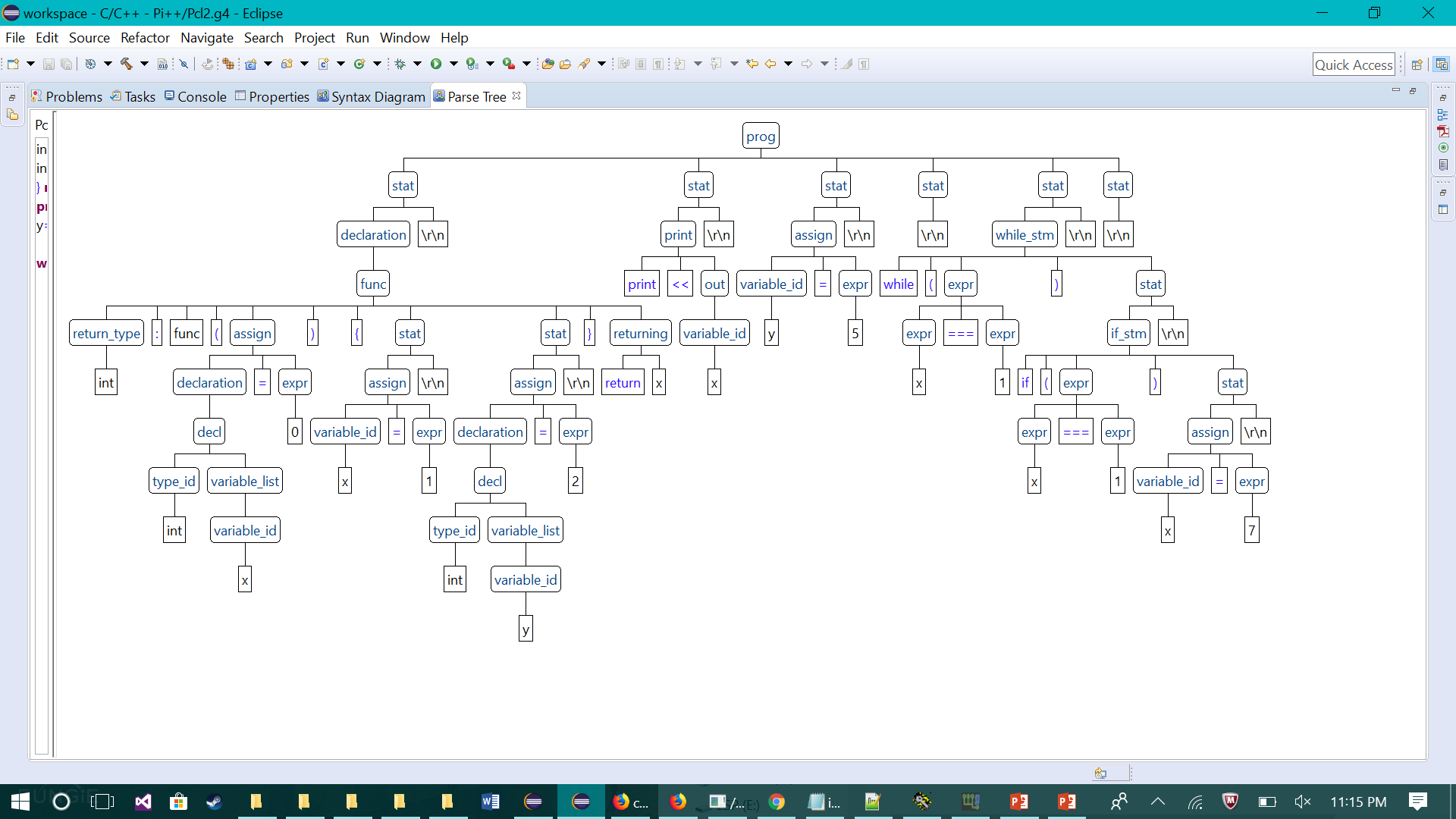
WS

The syntax diagram is a representation of our grammar file and how function and expressions are being executed as the program run. Its a path/rules that the program follows in order to execute.

For example an if statement is if(x== 2) y = 2 , the syntax diagram is going to be as follows.



**Parse Tree**

****

The above parse tree shows how antlr parses the sample program, according to the grammar file that was written with antlr syntax. Antlr will follow the syntax assigned in the grammar file and will generate a parse tree from its built in parser. As seen in the parse tree above stat is the start of the node then “assign” is the reserved word for assignment.Analysing the first branch of the parse tree the compiler is reading that the program is assigning a declaration of type id to a variable x and all this is being assigned to an expression. The compiler follows the parse tree and interpretes what the program wants to do and understands it, because it can interpret the syntax. The parse tree also shows the space “\r\n” before the parser goes to the next node it shows that the program creates space by executing “\r\n”.

The lexer/scanner from antlr while reading the files generates tokens, which are the passed to the antlr parser which generates the parse tree. Using the the visitor class antlr is able to generate a jasmin file which can be assembled to a .class and can be run on a java virtual machine