JARPL(Just A Random Programming Language)

Project Milestone 1

Project Members:
Saloni Chudgar - 1219586082
Sanjana Mukundan - 1219492846
Chandra Sekhar Sai Sampath Swaroop Atkuri -1219630568
Sudhanva Hanumanth Rao - 1219246938

Language Design :

Data Type	Bool (true or false) ,int (Integers)
Assignment Operator	=
Logical Operator	&& , | | , !!
Conditional Operator	if (expr) {code block} else {code block}
Arithmetic Operator	+, -, *, /, ^, %
Ternary Operator	<expr> ? <code block> : <code block>
Comparison Operators	==, >, <, <=, >=, !=

Iteration	while(expr){code block}
	
	for(<declaration>; <condition>; <increment>) { code block }
	
	for(<declaration> in range (<initial value>, <final value>) { code block }

Standard output	print <expr>;
Reserved Keywords	bool, int, for, if, else, while, in, range, print, true, false, main
Identifier	Must begin with a non-integer character
Symbols	;, {, } ,(,)

Logical Operators:

Operator	Name	Operands	Logical Operation
&&	Logical AND	a&&b	Both expression a and b must be true
||	Logical OR	a||b	Either of expression a or b must be true
!!	Logical NOT	!!a	Negation of a

Assignment Operators:

Operator	Name	Operands	Arithmetic Operation
+	Addition	a+b	Value of b is added to value of a
-	Subtraction	a-b	Value of b is subtracted from value of a
*	Multiplication	a*b	Value of a is multiplied with value of a
/	Division	a/b	Value of a is divided by value of b
^	Power	a^b	Value of a raised to the value of b
%	Modulus	a%b	Remainder when a is divided by b

Comparison Operators:

Operator	Name	Operands	Conditional Evaluation
==	Is equal to	a == b	Checks if b is equal to a
>	Is greater than	a > b	Checks if a is greater than b
<	Is less than	a < b	Checks if a is less than b
<=	Is less than or equals to	a <= b	Checks if a is less than or equal to b
>=	Greater than or equal to	a >= b	Checks if the value of a is greater than or equal to value of b
!=	Is not equal to	a != b	Checks if b is not equal to a

Ternary Operator:

Operator	Name	Operands	Ternary Evaluation
? :	Ternary Operator	<expr> ? a : b	If expression is true, execute a, else execute b

Language grammar :

<program> := main‘(’ ‘)’ ‘{‘ <block> ‘}’

<block> := <block><statement>
| ε

<statement> := <expr>;
| <declaration>;
| <while_cond>
| <if_cond>
| ‘{‘ <block> ‘}’

<expr> := <identifier>
| <assign_expr>
| <arith_expr>
| <comp_expr>
| <logic_expr>
| <tern_expr>

<declaration> := <data_type><identifier>
| <data_type><assign_expr>

<while_cond> := while ‘(‘ <expr> ‘)’ <block>

<if_cond> := if ‘(‘ <expr> ‘)’ <block>
| if ‘(‘ <expr> ‘)’ <block> ‘else’ <block>

<identifier> := <non_interger>
| <identifier><non_integer>
| <identifier><integer>

<assign_expr> := <identifier> ‘=’ <expr>

<arith_expr> := <arith_expr> ‘+’ <arith_expr>
| <arith_expr> ‘-’ <arith_expr>
| <arith_expr> ‘*’ <arith_expr>
| <arith_expr> ‘/’ <arith_expr>
| <arith_expr> ‘%’ <arith_expr>
| <arith_expr> ‘^’ <arith_expr>
| <integer>
| <non_integer>

<comp_expr> := <comp_phrase><comp_operator><comp_phrase>

<comp_phrase> := <identifier>
| <number>
| ‘(’ <arith_expr> ‘)’

<comp_operator> := ‘<’ | ‘<=’ | ‘>’ | ‘>=’ | ‘==’ | ‘!=’

<logic_expr> := <logic_phrase><logic_operator><logic_phrase>
<logic_phrase> := <identifier>
| <number>
| ‘(’ <arith_expr> ‘)’

<logic_operator> := ‘&&’ | ‘||’ | ‘!!’

<tern_expr> := <expr> ‘?’ <block> ‘:’ <block>

<number> := <number> <integer>
| <integer>

<data_type> := int | bool

<integer> := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<non_integer> := a | b | c | d .....x | y | z

<bool_val> := true | false

Parsing Technique to be used :

We are planning to use ANTLR for the parsing as well as to generate tokens(i.e as a lexer). The designed grammar file will be used as an input to the ANTLR.
ANTLR will generate two separate files based on the grammar which can be used a Lexer and Parser.
The Lexer will be used to convert the input into tokens which will then be utilized by the Parser to generate a parse tree.

Data Structures to be used:

Trees: Tree data structure will be used by the parser to create the parse trees. The parser will convert the tokens outputted by the lexer to a parse tree.

Stack: Our current plan is to use the stack data structure in our intermediate code generation. The parse tree from the parser will be converted to a stack format. However, should any issues arise, we will make need based changes.