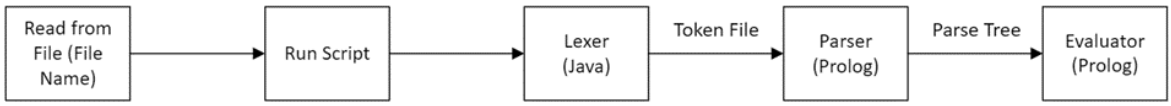
Name: Nimbus

Design:



Grammar:

```
program --> block, ['.'].

block --> ['main'], ['('], [')'], ['{'], decl_block.
decl_block --> decl, [';'].

%Predicate to parse the block
general_block --> ['{'], command, ['}'].

%Declarations(initialization)
decl -->  ['int'] ,identifier ,[':='] ,integer, [';'], decl.
decl -->  ['float'] ,identifier ,[':='] ,float, [';'], decl.
decl -->  ['string'] ,identifier ,[':='] ,string, [';'], decl.
decl -->  ['boolean'] ,identifier ,[':='] ,boolean, [';'], decl.
decl -->  ['int'], identifier.
decl -->  ['float'], identifier.

%Commands
command --> commandblock, [';'], command.
command --> commandblock.

%Command block(Loops, assignment, print commands)
commandblock --> ['if'], ['('], boolean,  [')'], general_block.
commandblock --> ['if'], ['('], boolean,  [')'], general_block,
['else'], general_block.
```

```
commandblock --> ['while'], ['('], boolean, [')'], general_block.
commandblock --> ['for'], ['('], ['int'] ,identifier ,['='] ,integer, [';'], boolean, [';'], unaryexpr, [')'], general_block.
commandblock --> ['print'], ['('], identifier, [')'].
commandblock --> general_block.

commandblock --> ['if'], ['('], boolean, [')'], general_block , [';'], command.
commandblock --> ['if'], ['('], boolean, [')'], general_block, ['else'], general_block, [';'], command.
commandblock --> ['while'], ['('], boolean, [')'], general_block, [';'], command.
commandblock --> ['for'], ['('], ['int'], identifier, ['='], integer, [';'], boolean, [';'], unaryexpr, [')'], general_block, [';'], command.
commandblock --> ['print'], ['('], identifier, [')'].
commandblock --> general_block.

commandblock --> identifier, [':='], expr.

%Increment decrement operators
unaryexpr --> identifier, ['+'], ['+'].
unaryexpr --> identifier, ['-'], ['-'].

ternaryexpr --> ['('], boolean, [')'], ['?'], expr, [':'], expr.

%Expression for arithmetic computations
expr --> factor.
expr --> miniexpr.
expr --> miniexpr, ['+'], expr.
expr --> factor, ['+'], miniexpr.
expr --> miniexpr, ['-'], expr.
expr --> factor, ['-'], miniexpr.
expr --> ternaryexpr.

%To incorporate precedence rules
miniexpr --> factor.
miniexpr --> factor, ['*'], miniexpr.
miniexpr --> factor, ['/'], miniexpr.
```

```
% Lowest factor in th expressions
factor --> identifier.
factor --> number.

%Boolean expressions
boolean --> ['true'] | ['false'] | ['not'], boolean.
boolean --> expr, ['='], expr , ['and'], boolean | expr, ['='],
expr, ['or'], boolean.
boolean --> expr, ['='], expr.

% identifier variables
identifier --> lowercase_letters, identifier.
identifier --> lowercase_letters, uppercase_letters, identifier.
identifier --> lowercase_letters, ['_'], identifier.
identifier --> lowercase_letters.

% String part
lowercase_letters --> ['a'] | ['b'] | ['c'] | ['d'] | ['e'] |
['f'] |
     ['g'] | ['h'] | ['i'] | ['j'] | ['k'] | ['l'] | ['m'] |
['n'] | ['o'] |
     ['p'] | ['q'] | ['r'] | ['s'] | ['t'] | ['u'] | ['v'] |
['w'] | ['x'] |
    ['x'] | ['y'] | ['z'].
uppercase_letters --> ['A'] | ['B'] | ['C'] | ['D'] | ['E'] |
['F'] |
     ['G'] | ['H'] | ['I'] | ['J'] | ['K'] | ['L'] | ['M'] |
['N'] | ['O'] |
     ['P'] | ['Q'] | ['R'] | ['S'] | ['T'] | ['U'] | ['V'] |
['W'] | ['X'] |
    ['X'] | ['Y'] | ['Z'].

% string part
string --> lowercase_letters, string.
string --> uppercase_letters, string.
string --> number, string.
string --> lowercase_letters.
string --> uppercase_letters.
string --> number.
```

```
% Numbers part
integer --> number, integer.
integer --> number.
float --> integer, ['.'], integer.
float --> integer.

number --> ['0'] | ['1'] | ['2'] | ['3'] | ['4'] | ['5'] | ['6']
| ['7'] | ['8'] | ['9'].
```

File extension for our language: “filename.nimb”

Why did we choose DCG to implement our parser?

For our grammar, we are using Definite Clause Grammar (DCG). It works well for small-scale projects if grammar rules are clear-cut and simple to follow. DCG offers flexibility and extensibility making it easier for us to modify overtime since we may not be sure of everything at the start of our project. Another reason for using DCG is because we are learning Prolog in this course, and we would like to apply what we have learned to build the grammar. While we compared our other options such as Yacc and ANTLR, we found that DCG was most well suited for our purpose because our grammar is not too complex. Moreover, DCG is simpler to learn than the others. In Yacc and ANTLR, the grammatical rules might grow intricate and challenging to adhere to. Additionally, they may be used for generating parses for domain specific languages.

Parsing technique and/or tools:

As part of the development of compiler and virtual machine we are using Java programming language to implement Lexer and Prolog language for our Parser and Evaluator. In milestone 2, if needed, we will be making use of the different open source and freely available tools to develop and build our project.

Data structures: List, Map

To Do for Final Milestone: Semantics for our language. List and Map implementation.