HABESOME

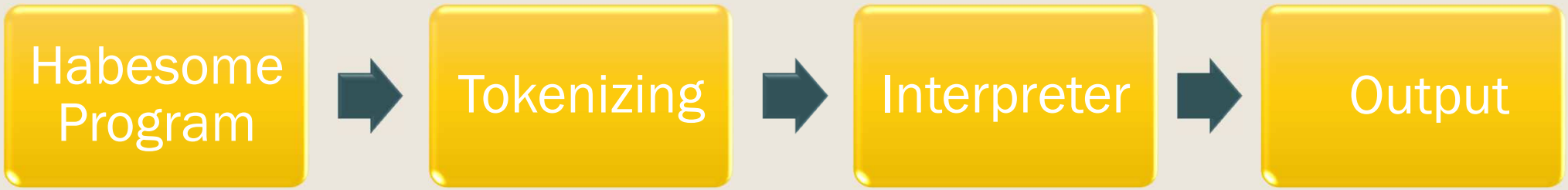
SER 502: Team 18

1) Melissa Day
2) Sowmya Madabhushi
3) Harshita Kajal
4) Behnaz Sabbaghi

Overview:

- Habesome Introduction
- Grammar
- Lexical Analysis
- Parser
- Interpreter
- Sample Programs

Habesome Design Flow



Grammar Description:

- Tokens in grammar are broken down into 1) terminals 2) non terminals
- The grammar handles precedence.
- Rules are defined for the grammar are:

Grammar

```
P is Program
K is Block
D is Declaration
DT is data types
SL is Statement List
S is Statement
A is Assignment Statement
IF is If Statement
W is While Statement
B is Boolean Expression
E is Arithmetic Expression
EX is Multiplication or Division Arithmetic Expression
I is Identifier
N is Number
INT is integer data type
BOOL is boolean type
DG is Digit
L is Letter
RES is resolution of addition or subtraction
RESMD is resolution of multiplication or division
FACTOR is a part (factor) of a multiplication or division expression

P ::= 'start' K 'end'
K ::= '[' D SL ']' | // Removed extra period
        '[' SL ']'
D ::= DT I '.' D |              // Fixed for adding assignment
        DT I '.' |
        DT A '.' D |
        DT A '.'

DT ::= 'int' | 'bool'   // fixed/updated
```

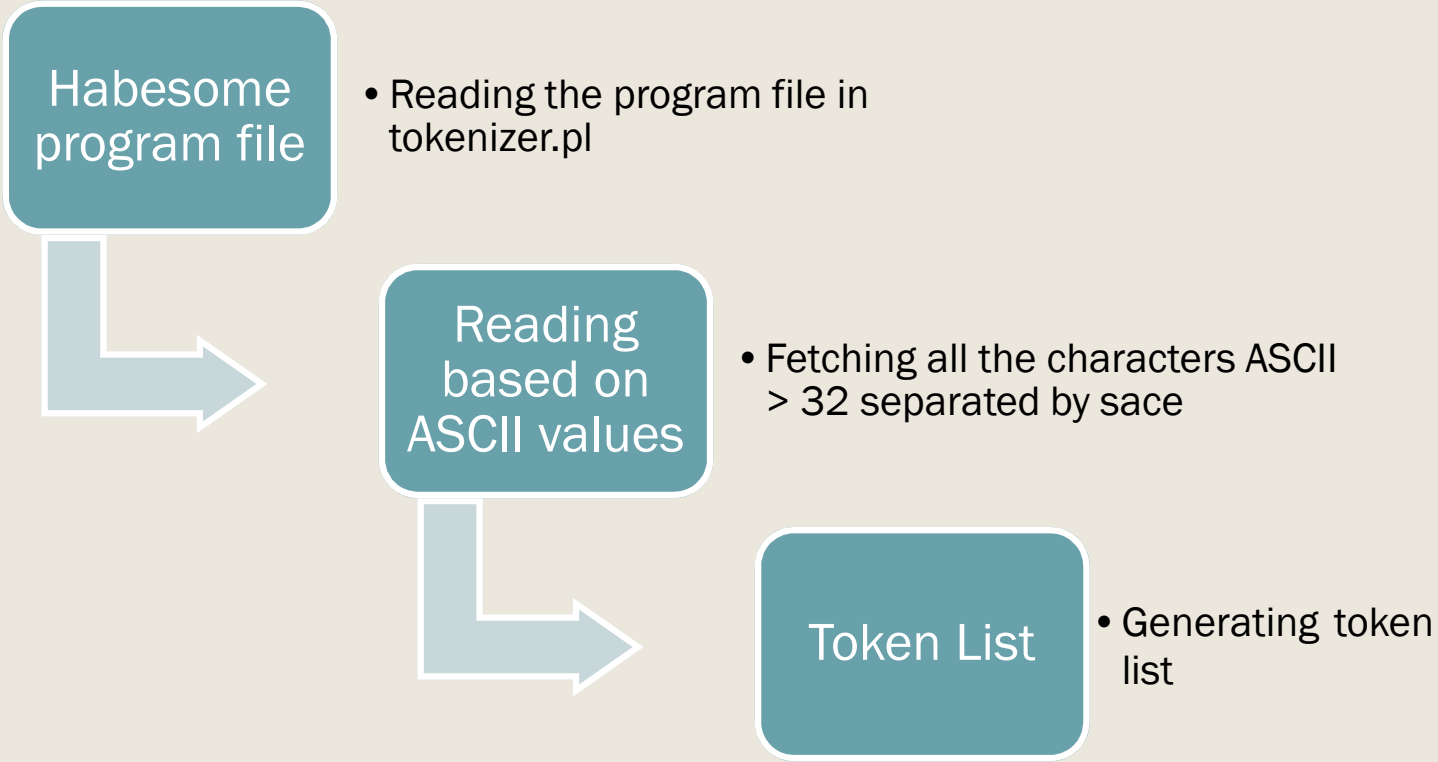
```

SL ::= S '.' SL |
        S '.' |
        K

S ::= A | IF | W

A ::= I 'is' E

IF ::= 'if' '(' B ')' K 'else' K

W ::= 'while' '(' B ')' K

B ::= 'true' |
        'false' |
        E '=:' E |
        '!!' E |
        E '<<' E |
        E '>>' E

==================================
// Left associativity fixed
E ::= EX RES |
                EX

RES ::= '+' EX RES |
                '-' EX RES |
                        '+' EX | '-' EX

// Left associativity fixed
EX ::= FACTOR RESMD |
                FACTOR
```

```
==================================
// Left associativity fixed
E ::= EX RES |
                EX

RES ::= '+' EX RES |
                '-' EX RES |
                        ‘+’ EX | ‘-‘ EX

// Left associativity fixed
EX ::= FACTOR RESMD |
                FACTOR

RESMD ::= '*' FACTOR RESMD |
                        '/' FACTOR RESMD |
                                        ‘*’ FACTOR | ‘/‘ FACTOR

FACTOR ::= I | N

==========================================

I ::= L I |
        L

N ::= DG N |
        DG

DG ::= [0-9]+

L ::= ('a'..'z' | 'A'..'Z')*
```

```
program :  block  ;
block : '[' ( declaration '.' statementList )']' |
   '[' statementList ']';
declaration : datatype statementList '.' declaration |
   datatype statementList '.';

   datatype: INT | BOOLEAN;

statementList : statement '.' statementList |
   statement '.' |
   block;

statement : assignment | ifStatement | whileloop;

assignment : identifier 'is' expression;

ifStatement : 'if' '(' bool ')' block 'else' block;

whileloop : 'while' '(' bool ')' block;

bool : 'True' |
   'False' |
   expression '=:' expression |
   '!!' expression |
   expression '<<' expression |
   expression '>>' expression;
```

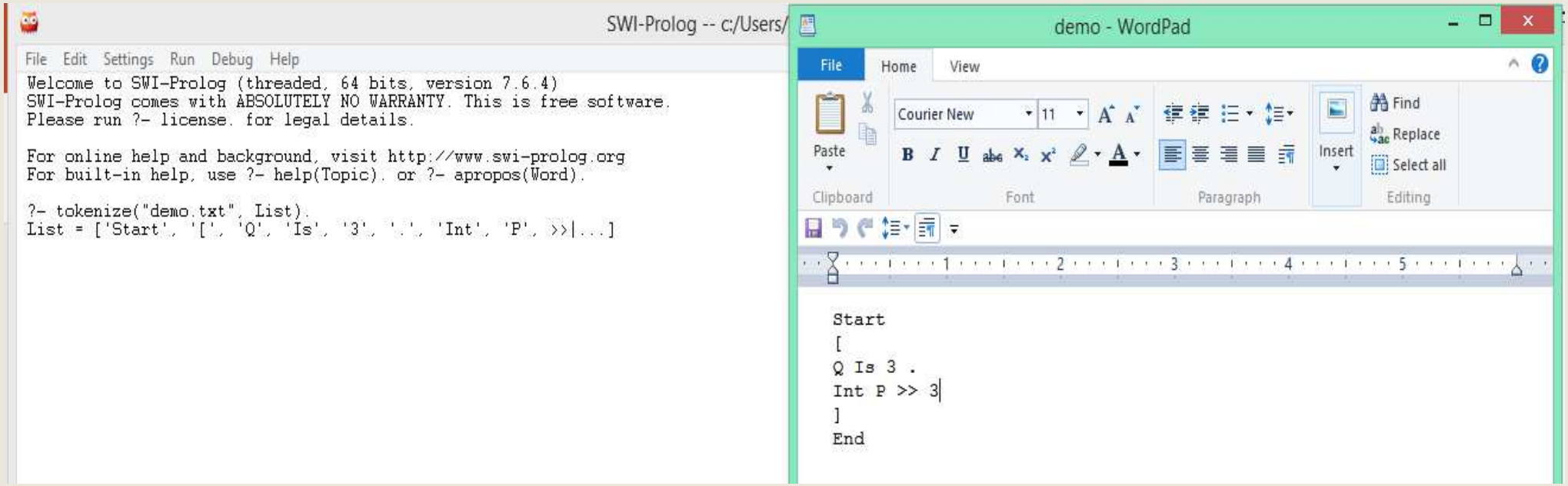
```
expression : term '+' expression |
   term '-' expression |
   term;

term : identifier '*' term |
   number '*' term |
   identifier '/' term |
   number '/' term |
   identifier |
   number ;

identifier: L identifier |
   L ;

number : DG number |
   DG ;
INT: [0-9]+ ;
DG : [0-9]+ ;
BOOLEAN: 'True' | 'False';

L : ('a'..'z' | 'A'..'Z')*;
```

Lexical Analysis



Sample output for tokenizer

- All the words in the file are separated by space
- Tokens generated are separated by comma



Parser

- Parser takes in the input generated from tokenizer
- Parse tree generated using a top-down approach
- The rules for parser are written in DCG (Definite Clause Grammar)

```
start
[
int p is 2.
q is 3.
]
end

% Execution -
?- L =['start','[','int','p','is','2','.','q','is','3','.',']','end'],program(P,L,[]).

% o/p -
L = [start, '[', int, p, (is), '2', ('.'), q, (is), '3', ('.'), ']', end],
P = parsetree(blockdec(deca(dtype1(int), assign(iden(letter(p)), arithexp(exp(factorn(num(digit('2')))))),
slist(stmtassign(assign(iden(letter(q)), arithexp(exp(factorn(num(digit('3')))))))))

% -
```

Interpreter

- Receives the parse tree as input from the parser
- Uses this predicate to run the interpreter

```
% Interpreter program predicate
    interpreter(PTokens, FinalEnv, OutputFileName) :-
        program(PTree, PTokens, []),
        writeOutputToNewFile(OutputFileName, "Parse Tree: "),
        writeOutputToExistingFile(OutputFileName, PTree),
        Env = [],
        evalProgram(PTree, Env, FinalEnv),
        writeOutputToExistingFile(OutputFileName, "Final Environment: "),
        writeOutputToExistingFile(OutputFileName, FinalEnv).
```

- Initializes the environment to an empty list
- Adds the new environment as pairs to the list

Writing a Habesome Program:

- Statements end with the period ‘.’ (fullstop)
- All the words in the program must be written with one character space (including the period and special characters like +, (,), etc ..)
- Assignment should be done for the same datatype
- Datatypes for a variable cannot be changed later in the program

Steps to run:

- The program should be saved as a .hbsm file
- In SWI-prolog we write the following execution query:

SWI-Prolog -- c:/Users/Melissa/Desktop/SER502_Repo/SER502-Spring2018-Team18/data/...

File Edit Settings Run Debug Help

```
Welcome to SWI-Prolog (threaded, 64 bits, version 7.4.1)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is fr
ee software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-pro
log.org
For built-in help, use ?- help(Topic). or ?- apropos(Wor
d).

?- main("Test5.hbsm",Output).
Output = [(a, 4),  (b, 3),  (c, 5)] .

?-
```

Steps to run:

- Program upon successful execution produces output in a .txt file format called “ProgramOutput.txt”
- File contains parse tree and output containing the final program environment

Steps to run:

- It generates an output file in the same directory with name “Program Output” which displays the required output and also the parse tree

```
% Runs the entire program from tokenizing to output.
% Saves results of parsetree and final environment to an output program called
% "ProgramOutput.txt"
% Expected Parameters:
% InputFileName: Filename where the program input is stored, Extension should be .hbsm

main(InputFileName, FinalEnvironment) :- tokenize(InputFileName, Tokens),
                                         interpreter(Tokens, FinalEnvironment, "ProgramOutput.txt").
```

Sample Program

```
start
[
if ( true )
[
d is 6 * 7 + 2 * 8 .
q is 3 .
] .
]
end
```

SWI-Prolog -- c:/Users/Melissa/Desktop/SER502_Repo/SER502-Spring2018-Team18/data/...

File Edit Settings Run Debug Help

```
Welcome to SWI-Prolog (threaded, 64 bits, version 7.4.1)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is fr
ee software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-pro
log.org
For built-in help, use ?- help(Topic). or ?- apropos(Wor
d).

?- main("Test4.hbsm",Output).
Output = [(d, 58),  (q, 3)] .

?-
```

Sample Program

```
Parse Tree:
parsetree(block(slist(stmtif(ifstate(boolexptrue(true),block(slistrec(stmtassign(assign(iden(letter(d)),
arithexp1(exp1(factorn(num(digit(6))),resmul2(factorn(num(digit(7))))),resarithsum(exp1(factorn(num(digit(2))),
resmul2(factorn(num(digit(8))))))))),slist(stmtassign
(assign(iden(letter(q)),arithexp(exp(factorn(num(digit(3)))))))))))))))
Final Environment:
[(d,58),(q,3)]
```