User Manual

1. Brief introduction: The language used to design compiler is Java.
2. Software requirements: Eclipse (any version), a MIPS simulator and a text editor.
3. Getting started: Load the project in Eclipse. Create a text file inside the project folder. The program can be written in any text editor. But it should be placed inside the source of Eclipse project. Next, run the project from the eclipse environment. If the output has no syntax error or warnings, a file named "output.s" will be generated. Load the output.s in MIPS assembler and then run it. This will generate the desired output.
4. List of Reserved Tokens: Listed below are some reserved tokens.

```
BEGIN               END               COMMENT
IF                  THEN              (*no ELSE in the language*)

WHILE               DO
INTEGER             STRING            LOGICAL                 REAL
TRUE                FALSE             DIV                     REM
OR                  AND
READ                WRITE             WRITELN
```

5. Some General guidelines:
    - The language is case sensitive for keywords . For instance, integer is different from INTEGER. Types are integer, real, string, and Boolean
    - The name of an identifier should start with a letter.
    - All the statements must be separated by a semi-colon ';'
    - There are 4 kinds of tokens, listed below
      punctuation: punctuators, infix operators, unary operator
      literals: character strings, integer literals, Boolean literals
      keywords: list is above
      identifiers
    - The symbol COMMENT followed by any sequence of characters not containing semicolons, followed by a semicolon, is called a comment.
    - All the variables should be declared first before initializing.
    - The body of the program should be between the Begin and end block and the body of the statements should be between the Begin and end block.
    - The only loop that Bnf supports is while. The same applies for if-else statements.

Technical Documentation

Known Bugs:

- The program is having trouble when implementing the code generation. Both findInAll and findInCurrent methods of the symbol table in code generation are working correctly. Just need few more days to debug for code generation.

Unimplemented Parts of the language

- Need to debug and implement the code generation.

List of tokens with token nums

```
 1            identifier
 2            any literal (integer such as 123,string "abc", FALSE,
              TRUE)

 3            types (keywords 'INTEGER' 'STRING' 'LOGICAL')
 4            addition operators (+  -  OR)
 5            multiplication operators (* / DIV  REM AND)
 6            relational operators = , !=, <, >
 7           BEGIN
 8           END
 9  IF
 10  THEN            (*no ELSE in the language*)
 11  WHILE
 12  DO
 #Token 2
 2   TRUE                  2   FALSE

 #Token 3
 3   INTEGER               3   STRING       3   LOGICAL

 #Token 4
 4   OR

 #Token 5
 5   DIV                   5  REM                      5   AND

 #Token 13
 13   READ                 13  WRITE        13  WRITELN
(IO_operations)
 14  (
```

```
15  )
16 ;  (not part of a comment)
17  !  Boolean not
18  .  (only at the end of the program)
19. =
```

A sample program

```
BEGIN        COMMENT SAMPLE PROBLEM;
INTEGER N;
REAL MAX, PRICE, SUM;  COMMENT we won't need the comma;
N := 0;
SUM := 0;
MAX := 0;
READ(PRICE);
WHILE PRICE > 0 DO
BEGIN
N := N + 1;
SUM := SUM + PRICE;
IF PRICE > MAX
THEN MAX := PRICE;
READ(PRICE);   COMMENT readon not included in tiny Algol-W;
END;
WRITE(N,SUM,MAX)
END.
```

BNF with 17 nonterminals

```
1.program               :  blockst '.'
2.blockst               :  BEGINTOK stats ENDTOK
3.stats                 :   statmt ';'  stats   | <empty>
4.decl                  :  BASICTYPETOK IDTOK
5.statmt          : decl | ifstat | assstat |  blockst | loopst |
iostat | <empty>
6.assstat           :  idref   ASTOK  expression
7.ifstat            :  IFTOK  expression THENTOK statmt
8.loopst            :  WHILETOK expression DOTOK stat
9.iostat            :  READTOK ( idref ) | WRITETOK (expression)

10.expression        :  term expprime
                        E → T E'
11.expprime          :  ADDOPTOK  term expprime|<empty>
                        E' → ADD T E'|eps
12.term              :  relfactor termprime
                         T → RF T'
13.termprime         :  MULOPTOK  relfactor termprime|<empty>
                        T'→ MUL RF T' | eps
14.relfactor         :  factor factorprime
                        RF → F F'
15.factorprime       :  RELOPTOK  factor          |  <empty>
                         F' → REL F  | eps
16.factor            :  NOTTOK  factor| idref | LITTOK
|'('expression')'
                        F → ID | LIT | NOTID | ( E )


17. idref            : IDTOK
```

Notes: 1. OR was combined with ADDOPs, AND combined with MULOPS to reduce the levels of precedence and shorten the grammar. This deviates from true Algol.

2. There is no unary – or +. Having them adds to the complexity for the scanner/parser. Someone has to know whether to treat each + or – as a binary operator or a unary operator. Example: a- -5 + -b.
For more info, see:

Future work

- Change the BNF grammar of the language to include identifiers to be passed as array references. Eg. A[i].
- Introduce some more tokens to get accepted by the Lexical Analyzer.
- Introduce Switch and case statement in BNF.
- Introduce for loop .