# Custom Language to Assembly Compiler

Program takes input either from file or from keyboard and outputs .asm files.

- Main file – Takes input and produces .asm file
- Scanner – Reads characters and produces tokens, checks for syntax errors
- Parser – Interprets tokens
- Semantics – Checks for semantic errors

Lexical Definitions

- Alphabet
  - all English letters (upper and lower), digits, plus the extra characters as seen below, plus WS
  - No other characters allowed and they should generate errors
- Identifiers
  - begin with a letter and
  - continue with any number of letters or digits
  - you may assume no identifier is longer than 8 characters (in testing)
- Keywords (reserved, suggested individual tokens)
  - ```
    begin end loop void var return in out program iffy then let data
    ```
- Operators and delimiters group (all single character except ==)
  - ```
    = < > == : + - * / % . ( ) , { } ; [ ]
    ```
- Integers
  - any sequence of decimal digits, no sign, no decimal point

Semantics

- Basic semantics as in C - program executes sequentially from the beginning to the end, one statement at a time
- Conditional statement is like the else-less if statement in C
- Loop statement is like the while loop in C
- Assignment evaluates the expression on the right and assigns to the ID on the left
- +-* are standard arithmetical, / is integer division, unary * is negation
- All expressions are evaluated before being used
- Relational and arithmetical operators have the standard meaning except:
  - < and > are less than and greater than
  - < < is less equal
  - > > is greater equal
  - < > is NOT equal
  - == is equal
- IO reads/prints a 2-byte signed integer
- All data is 2-byte signed integer

BNF

```
<program> ->   <vars> <block>
<block>   ->   begin <vars> <stats> end
<vars>    ->   empty | data Identifier = Integer . <vars>
<expr>    ->   <N> - <expr> | <N>
<N>       ->   <A> / <N> | <A> * <N> | <A>
<A>       ->   <M> + <A> | <M>
<M>       ->   * <M> | <R>
<R>       ->   ( <expr> ) | Identifier | Integer
<stats>   ->   <stat> <mStat>
<mStat>   ->   empty | <stat> <mStat>
<stat>    ->   <in> . | <out> . | <block> | <if> . | <loop> . | <assign> .
<in>      ->   in Identifier
<out>     ->   out <expr>
<if>      ->   iffy [ <expr> <RO> <expr> ] then <stat>
<loop>    ->   loop [ <expr> <RO> <expr> ] <stat>
<assign>  ->   Identifier = <expr>
<RO>      ->   < | < < (two tokens >) | > | > > (two tokens) | == (one token ==) | < > (two tokens)
```

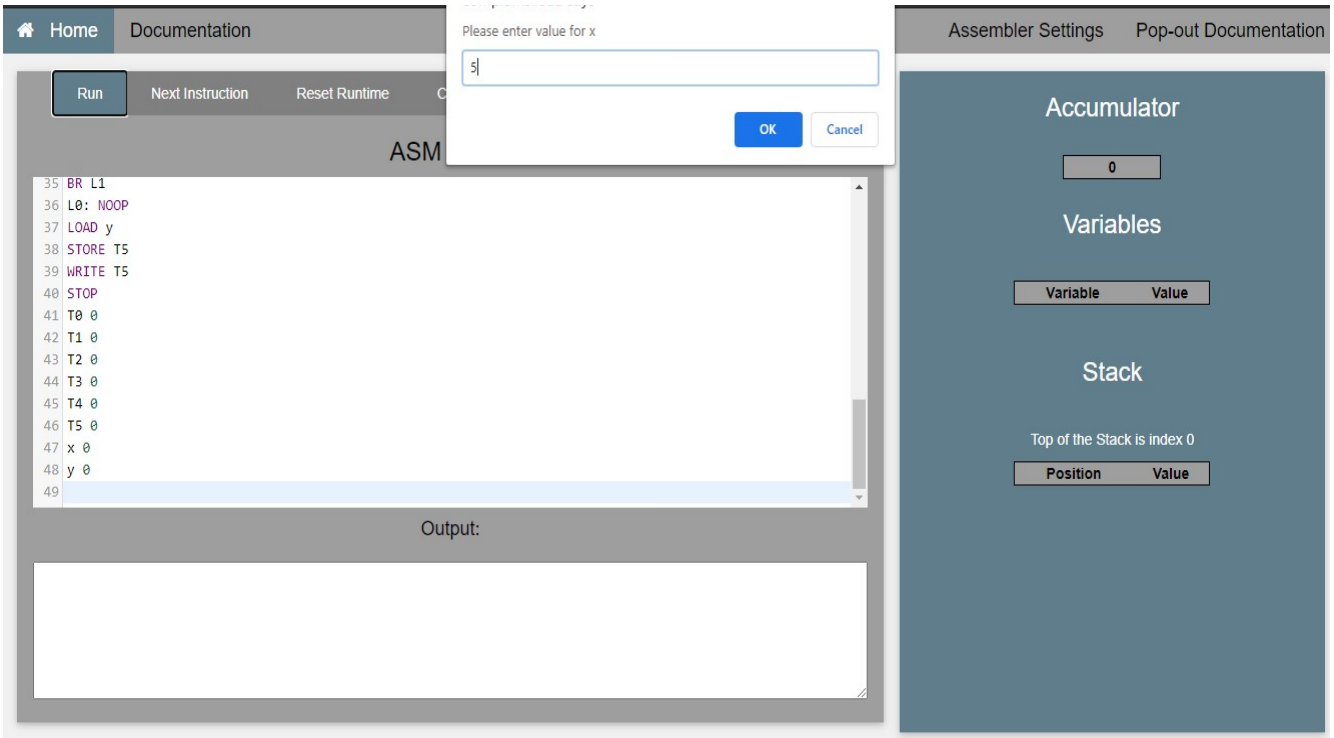
Example Input

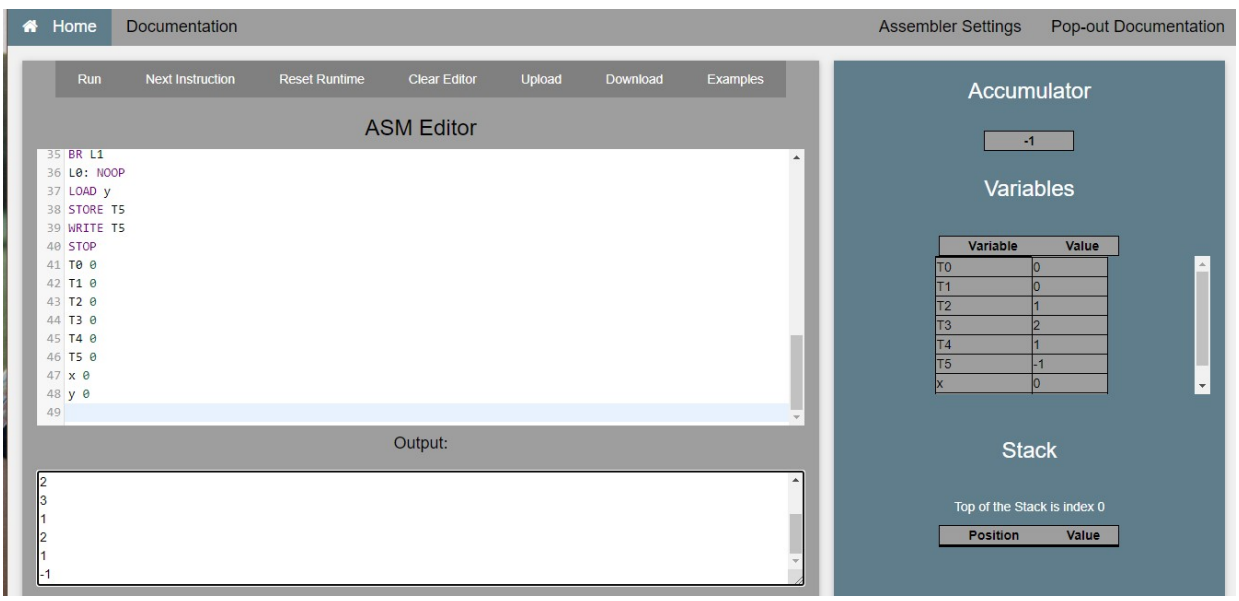
```
data x = 1 .
data y = 2 .
begin
        in x .
        loop [ x > 0 ]
        begin
                y = x .
                loop [ y > 0 ]
                begin
                        out y .
                        y = y - 2 .
                end .
                x = x - 1 .
        end .
out y .
end
```

Example Output

```
LOAD 1
STORE x
LOAD 2
STORE y
READ x
L1: NOOP
LOAD 0
STORE T0
LOAD x
SUB T0
BRZNEG L0
LOAD x
STORE y
L3: NOOP
LOAD 0
STORE T1
LOAD y
SUB T1
BRZNEG L2
LOAD y
STORE T2
WRITE T2
LOAD 2
```

```
STORE T3
LOAD y
SUB T3
STORE y
BR L3
L2: NOOP
LOAD 1
STORE T4
LOAD x
SUB T4
STORE x
BR L1
L0: NOOP
LOAD y
STORE T5
WRITE T5
STOP
T0 0
T1 0
T2 0
T3 0
T4 0
T5 0
x 0
y 0
```




Input 5

Output 5, 3, 1, 4, 2, 3, 1, 2, 1, -1