Why This Language:
Working with x86-64 assembly is a more challenging way to learn about how memory works. Therefore, this can also be applicable to writing code in C-like languages, such as with better understanding pointers. This is a modified version of the Lowlang Function Pointers from the example language proposals given. This is also intended to understand and explore the compiling of assembly on current and more modern platforms.

Code Snippets:

Example Program:

```
(extern MessageBox (int (*char) (*char) int) int)

    (func main () void

        (vardec (*char) title)
        (vardec (*char) message)

        (assign title ""Hello World"")
        (assign message ""Hello from BaD"")

        (call MessageBox 0 title message 0)
    ))
```

The above example is a simple “hello world” program. This includes a main function which declares and assigns the variables “title” and “message”, and then calls the specified function.

Known Limitations:
Among the limitations of the language, there is some undefined behavior, along with only having stack allocation built-in. This includes uninitialized values, accessing memory out of bounds, unaligned access, and invalid signatures on externs. Memory access issues covers cases such as

What would you do differently?:

Ideally, we would have liked to cover any limitations previously mentioned. However, this would have been much harder to tackle due to time constraints, along with needing more experience with x86. The biggest challenge was understanding code generation and understanding how to work with registers. Changing the testing process could also be more beneficial as to fully cover more problematic cases and potentially niche errors as the components are being written.

How to Compile and Run Compiler:

FASM (Flat Assembler) must be downloaded and installed in the same location as the program itself. The run.bat file will open the command prompt so that the user may run

Formal Syntax Definition:

`var` is a variable
`i` is an integer
`c` is any printable character

```
type ::= `int` | Integers are a signed type (64 bits)
          `int16` | Integers are a signed type (16
         bits)
          `int32` | Integers are a signed type (32 bits)
          `uint` | unsigned integers are a type (64 bits)
          `uint16` | unsigned integers are a type (16
         bits) `uint32` | unsigned integers are a type
         (32 bits) `void` |
        `char` | a signed character/number (8 bits)
        `uchar` | an unsigned character/number (8 bits)
        `(` `func` `(` type* `)` type `)` Function
       pointer `(` `*` type `)` pointer to type


param :: = `(` type var `)`

`
escapeCharacter ::= `\n` | `\t` | `\r` | `\0`

characterLiteral ::= `'` (c | escapeCharacter) `'`
stringLiteral ::= `"` (c | escapeCharacter)* `"`

Comments
```

```
comment ::= `//` c*
```

Functions

```
fdef ::= `(` `func` var `(` param* `)` type stmt* `)`
externdef ::= `(` `extern` var `(` type* `)` type stringLiteral
`)` | External function statement
`(` `extern` var `(` `...` `)` type stringLiteral `)` |
variadic function, only valid for extern

      stmt ::= `(` `vardec` type var `)` | Variable
    declaration `(` `assign` var exp `)` | Assignment
        `(` `assignptr` var exp `)` | Move a value into
the memory that a pointer points to
         `(` `while` exp stmt `)` | While loops
         `(` `if` exp stmt [stmt] `)` | if
         `(` `return` [exp] `)` | Return
         `(` `block` stmt* `)` | Blocks
```

Arithmetic and relational operators

```
op ::= `+` | `-` | `*` | `/` | `<` | `>` | `<=` | `>=` | `==` |
`!=`

exp ::= i | `true` | `false` | Integers, booleans
        var | Variables
         `null` | Null; assignable to pointer types
        characterLiteral| Character literal, result is char
stringLiteral | String Literal, result is (*char) to memory
of a null terminated string of characters `(` `array` exp
exp*`)` | Making a stack allocated array, result is (*type),
where type is type of the first expression (exp). ex. (array
1 2 3 4)
         `(` `cast` type exp`)` | Cast a value from one type
to another
         `(` `&` var `)` | Getting the address of a
variable/function
         `(` `@` var `)` | Dereference a pointer
         `(` op exp exp `)` |

        Function call. The first exp will either be a function
        name, or will evaluate to a function pointer. The
        typechecker must disambiguate between the two.
         `(` `call` exp exp* `)`
          `(` `index` exp exp `)` | Index a pointer (index *
sizeof(type) + exp2), does not dereference
          `(` `sizeof` (var|type) `)` | Size of a
variable/type, result is a uint64
```

```
maindef ::= `(` `func` `main` `(` `(` `int` var `)` `(` `*` `(` `*` `char` var `)` `)` `)` `int` stmt* `)` stmt is the entry point
program ::= (fdef | externdef)* maindef
```