README
Semantic Analyzer
Dylan Miller

This Parser builds on top of the Lexical Analyzer that had been previously designed. Once the Parser is setup by the driver, it requests tokens one at a time from the Lexical Analyzer. It uses a static Token that exists in Parser.c to pass the token back and forth from the analyzer. One the token is received, the Parser functions via a mutual recursive descent model which passes the token through the functions in accordance with the grammar that has been designed for Morbius17. Each non-terminal in the grammar has its own function in the Parser. The left hand side of the production is apparent by the function name, and the body of the function is determined solely based upon the demands of the particular production for the given non-terminal. The result is that the descent through the series of recursive functions acts analogous to an expansion of each production in the manner that the language grammar prescribes.

The semantic analyzer extends the recursive descent parser of the compiler. To ensure that the source files make semantic sense, the analyzer establishes scopes in in the source programs and defines anything declared as being attached to whichever scope it is declared in. Whenever a variable or other declarable “thing” (i.e. struct, function, parameter, etc…) is referred to in the source code, the semantic analyzer checks the current scope to see if the declaration exists within it. If not, the analyzer checks back through successive scopes until it either finds the declaration or exhausts its search. It passes information during the declaration process by passing type information between the recursive functions so that correct types and modifiers can be attached to their given instances. The output for the semantic analyzer tells when a declaration is being made and when scopes are being entered or exited.

The code generation portion build on top of the semantic analyzer. It passes the LocGen struct, which captures the location of the new variable/parameter/struct member to be declared, through declarative paths in the parser.c file so that it can be stored in the DeclarationList for its given scope. These locations can then be later referenced in order to generate assembly code that pulls from the locations at which the variables are stored. It uses the generate() function in order to create a line of code in the program’s output.
To collect the declarations to be used in the expressions (or for another purpose) the parser passes instances of the Result struct, which gets the location and can determine if the declaration is to be used for a left hand side, right hand side, or some other way. Two instances of the struct are created/passed between functions: the one containing the left hand side of the expression and the one carrying the output from its use. If the left hand side of the expression is the only part of the expression, the output gets the left hand side and it is recursively passed back up through the calling functions.

To use the compiler:

```
make
make test
```

make test will run ./m17 test_0/test3.m17