MDB Compiler User Manual

Matthew Benson

Version 1.02

August 16, 2016

Table of Contents

[List of Tables - 2 -](#_Toc459396665)

[Introduction - 3 -](#_Toc459396666)

[Chapters - 3 -](#_Toc459396667)

[How to start - 3 -](#_Toc459396668)

[Data and Data Types - 4 -](#_Toc459396669)

[Operators - 5 -](#_Toc459396670)

[Conditional Statements - 5 -](#_Toc459396671)

[Appendices - 6 -](#_Toc459396672)

[Syntax Diagram - 6 -](#_Toc459396673)

[Error Messages - 7 -](#_Toc459396674)

# List of Tables

[Syntax Diagram - 6 -](#_Toc459395962)

[Error Messages - 7 -](#_Toc459395963)

# Introduction

MDB Compiler (Matthew David Benson) is a free, open source compiler written in the Java programming language for COSC 470 at Frostburg State University. The MDB Compiler supports both character and integer data types, and has multiple running options from the command line to change the displayed output or automatically run the program when the compiler is finished.

A few unique features of the compiler are the ability to store numbers into a char variable and have it store the ASCII value in that variable and the potential to produced moderately optimized codes for most inputs.

# Chapters

## How to start

The MDB Compiler can be ran by navigating to CompilerDistributable/ on the command line then run the program with:

‘java -jar MDBCompiler.jar <source file> <options>’

Running the program with no arguments will display a help message about running the program and the meanings of the different arguments. The name or path of any plain text file needs to be passed to the compiler in the <source file> spot immediately following MDBCompiler.jar, which is the source code target of the compiler. A possible configuration for running a sample file from the CompilerDistributable folder would look like this:

‘java -jar MDBCompiler.jar SampleFiles/ConditionalInput’

There are three command line options for the compiler that can be added at the end in any order. The possible command line options are as follows:

-v Output messages about what the compiler is doing

-r Runs the output file if compiling is successful

-h or -help Display this message

One possible configuration for running the compiler with multiple options is this:

‘java -jar MDBCompiler.jar SampleFiles/ConditionalInput -r –v’

The -v option makes the compiler output information about the parse table states, the parse stack, and the productions used when reducing. The intermediate code file will be created and then read by Mini to create the executable file. The -r option now causes mICE to run with the code created by Mini.

## Data and Data Types

The MDB Compiler supports int and char data types. Variables must be declared before they can be used elsewhere in the program. To create a variable, declare it with:

‘var <name>: <type>;’

To declare multiple variables of the same type, they can be compacted to one line like this:

‘var <name1>, <name2>, <name3>: <type>;’

The two possible options for <type> are ‘int’ and ‘char’ and it should be noted that int variables cannot hold char values. While it is not possible to store every possible character with the format ‘x’, it is possible to store a number into a char variable. This allows any ASCII character to be stored easily.

There are no naming conventions enforced by the compiler, but reserved words cannot be used for variable names. Once a variable is declared, it can be used to store data of the declared type, hold the result of an arithmetic expression, or hold data to be printed with ‘put(variable).’

## Operators

The basic arithmetic operators are allowed in the input file. The syntax for expressions can be observed in the Syntax Diagram section on page 6. Generally expression use this syntax:

<varName> = <expression> ;

Where <expression> could be nearly any arithmetic such as ‘5 + 6’ or ‘var1 \* var2 + 4’ or any other combination of terms and operators that reduce to an *expression* in the Syntax Diagram. The valid operators that make up <expression> are ‘+’, ‘-’, ‘\*’, ‘/’, and ‘%’. It should be noted that not any possible arithmetic expression is accepted in this language. The expression ‘4 + var1 \* var2’ will not be accepted because when it reduces, it goes to ‘simple\_expression addop term \* var2’ then to ‘simple\_expression \* term’ and finally to ‘simple\_expression mulop term’ which has no way of reducing and is rejected by the parse table.

## Conditional Statements

Conditional statements in the form ‘*if ( expression ) statement*’ are accepted by the MDB Compiler where the expression is usually one that contains a relationship operator, although there are a few additional cases where different expressions are accepted. For instance, using ‘true’ for the expression will cause that code to always execute, while putting ‘false’ will create dead code that will never execute. The allowed relationship operators are ‘<’, ‘<=’, ‘==’, ‘>=’, ‘>’, and ‘<>’. The ConditionalInput sample file distributed with this documentation contains several examples of relation operators and can be helpful in understand their functionality.

# Appendices

## Syntax Diagram

1. start -> access static void ID ( identifier\_list ) { declarations compound\_statement } $
2. access -> public
3. access -> private
4. identifier\_list -> ID
5. identifier\_list -> identifier\_list , ID
6. identifier\_list -> [null]
7. declarations -> declarations var identifier\_list : type ;
8. declarations -> [null]
9. type -> char
10. type -> int
11. compound\_statement -> { statement\_list }
12. statement\_list -> statement
13. statement\_list -> statement\_list ; statement
14. statement -> lefthandside
15. statement -> compound\_statement
16. statement -> get ( ID )
17. statement -> put ( ID )
18. statement -> if ( expression ) statement
19. lefthandside -> ID = righthandside
20. righthandside -> expression
21. expression -> simple\_expression
22. expression -> simple\_expression relop simple\_expression
23. simple\_expression -> term
24. simple\_expression -> simple\_expression addop term
25. term -> factor
26. term -> term mulop factor
27. factor -> ID
28. factor -> num
29. factor -> true
30. factor -> false
31. factor -> ' literal '
32. relop -> >
33. relop -> >=
34. relop -> ==
35. relop -> <=
36. relop -> <
37. relop -> <>
38. addop -> +
39. addop -> -
40. mulop -> \*
41. mulop -> /
42. mulop -> %

## Error Messages

1. File Not Found – “Unable to open file: <”fileName”>” – This error occurs when the file passed to the compiler either doesn’t exist or is locked by the operating system or another program.
2. Cannot Parse – “Unable to parse input file: make sure to include file end character $” – This error occurs if the end of the file is not reached when scanning the file for tokens. It is usually caused by omitting ‘$’ at the end of the file.
3. Syntax Error – “Input is rejected - unable to compile while parsing. Check syntax of input and try again.” – This error is triggered if the parse table goes to a rejecting state while reducing.
4. Reduction Error – “Encountered an error when reducing - could be a type mismatch or another gramatical or logical error” – This generic error message is triggered if the parser would fail to reduce for any reason.
5. Name Taken – “Error: Symbol already in table - <name>” – This error will display if two variables are declared with the same name.
6. Name Does Not Exist – “Error: Name does not exist in symbol table: <name>” This error will display if a name is called in another operation but that name has not been declared as a variable.
7. Failure At Token – “Failure at token: <TokenName>, <TokenNumber>” – This error is displayed if a token is invalid and causes the parse table to go to a rejecting state.
8. Failure To Create Executable – “Failure in mini: unable to create executable file” – This error is displayed if Mini has any failure when running.
9. Failure To Execute – “Failure in mice: unable to execute file” – This error is displayed if mICE has any failure when running.