MDB Compiler System Manual

Matthew Benson

Version 1.02

August 16, 2016

Contents

[List of Tables - 2 -](#_Toc459581447)

[Introduction - 3 -](#_Toc459581448)

[Chapters - 3 -](#_Toc459581449)

[Grammar - 3 -](#_Toc459581450)

[Backus-Naur Form Syntax Diagram - 3 -](#_Toc459581451)

[Symbols - 4 -](#_Toc459581452)

[Operators - 5 -](#_Toc459581453)

[Parser - 6 -](#_Toc459581454)

[Symbol Table - 7 -](#_Toc459581455)

[Lexical Analyzer - 7 -](#_Toc459581456)

[Intermediate Code Generation - 8 -](#_Toc459581457)

[Outcome of unexpected termination - 9 -](#_Toc459581458)

[Appendix - 10 -](#_Toc459581459)

[Program Examples - 10 -](#_Toc459581460)

# List of Tables

[Backus-Naur Form Syntax Diagram - 3 -](#_Backus-Naur_Form_Syntax)

[Symbols - 4 -](#_Toc459581067)

[Relation Operators - 9 -](#Relop)

[Program Examples - 10 -](#_Toc459581075)

# 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

## Grammar

### Backus-Naur Form 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 -> %

### Symbols

The terminal and non-terminal symbols are listed below with their corresponding number that represents them in the compiler and parse table. Terminal symbols range from zero to thirty-seven and non-terminals range from 100 to 116. The uppercase terminal symbols represent raw data values such as a number (1, 2, 3, etc.) or a character (‘a’, ‘b’, ‘c’, etc.)

1. $accept
2. static
3. void
4. ID
5. public
6. private
7. var
8. :
9. ;
10. {
11. }
12. (
13. )
14. $
15. char
16. int
17. get
18. put
19. if
20. =
21. NUM
22. true
23. false
24. >
25. >=
26. ==
27. <=
28. <
29. <>
30. +
31. –
32. \*
33. /
34. %
35. LITERAL
36. $end
37. ,
38. ‘
39. error
40. start
41. access
42. identifier\_list
43. declarations
44. type
45. compound\_statement
46. statement\_list
47. statement
48. lefthandside
49. righthandside
50. expression
51. simple\_expression
52. term
53. factor
54. relop
55. addop
56. mulop

### Operators

The operators allowed by the MDB Compiler are those that reduce to either relop, addop, or mulop. The addop operators are ‘+’ and ‘-‘, the mulop operators are ‘\*’, ‘/’, and ‘%’, and the relop operators are ‘<’, ‘<=’, ‘==’, ‘>=’, ‘>’, and ‘<>’. For more information on the use of operators in the compiler, see the Lexical Analyzer and Intermediate Code section of the manual.

## Parser

The MDB Compiler uses a top-down parser with an LL(1) grammar, meaning the scanner and parser go from the top of the file to the bottom and read the lines from left to right, while looking at one token at a time. This is not the most powerful parser, but it is simpler to implement and follow.

Three tables are used to keep track of the rules of the language for the parser. There is a parse table, a grammar table, and a production mapping table. The parse table used is stored in the Parser class and stores the action table as well as the go-to commands in one two-dimensional array called parseTable. It may be difficult to read the parse table from the Java source file, so an excel spreadsheet with the parse table is included with this documentation.

The grammar table is also stored in a two-dimensional array in the Parser class but the format is much different. The first element is the non-terminal which produces the elements on the right side which can be found in the syntax diagram above. The second element shows how many additional elements are in the row, which saves time when figuring out how many nodes to pop off during reducing. After the second element is the list of symbols for that production.

The production mapping table is also stored in the Parser class, and it only has one purpose: to match symbol numbers to their column in the parse table. The parse table is not laid out in a way that reflects the numbering of possible symbols, so this HashMap called productionMapping associates column numbers with symbol numbers and is used to help with parsing.

The parse stack does not have any size limitations since it is implemented with java.util.ArrayDeque<ParseStackNode> which grows as necessary to support any amount of nodes. There are other errors that could be thrown by the compiler such as reduction errors and reaching rejecting states. For more details on error messages, consult the user manual.

## Symbol Table

The symbol table used by the MDB Compiler implements a Hash Table using a java.util.LinkedList<Symbol>[] object. This means that it is an array of type LinkedList<Symbol>, so there is no size limitation to the symbol table, since the LinkedList ensures to expand itself to fit any amount of data. The symbol table implementation uses open hashing, meaning if there is a collision in the table, those elements are still stored in the same LinkedList. The hash function used to choose an index for a symbol that is inserted to the table is java.lang.String.hashCode() and it uses the String name of the symbol being inserted. The SymbolTable class also has accessor and setter methods for the different data elements of a symbol stored in the table, and also has a toString() method that utilizes Symbol.toString() to print a well formatted table. For more information, see the source code’s Javadoc. The symbol table is also responsible for throwing any errors related to duplicate or missing symbol names.

## Lexical Analyzer

The Lexical Analyzer in the MDB Compiler is mostly implemented with the FileScanner class, which breaks input Strings from the source file into lexemes and eventually creates the tokens that are given to the parser. This class contains several public interface methods which are used to select a file to scan, start scanning, and get the next token, but the heavy work is handled in the private method scanHelper(). Essentially the method has a list of separators that it knows to identify separate lexemes with, but also checks for separators that are two characters such as ‘<=’ as well as a few other cases. For the fine details, see the Javadoc comments in the source file for the FileScanner class.

## Intermediate Code Generation

All of the intermediate code generation done by the MDB Compiler is done in the Parser class in the method generateCodeFromProduction(). The basic algorithm for this method is as follows: get the production being used from the syntax diagram, go to the rule for that production number, execute that statement, return.

For variable declarations, every time there is a variable name that reduces to ‘identifier\_list’ (note: ignores the first identifier\_list since it is the parameter list), that variable is added to a stack of declarations, then when the entire declaration is reduced to ‘declarations var identifier\_list : type ;’ the Strings in the declaration stack are popped off and given the type found when reducing, then added to the symbol table with a value that is incremented as variables are added in order to keep track of the variable in the intermediate code.

For assignment statements, code is generated to evaluate the right side of the statement then store it in location zero, which is our working space. Once all code needed to evaluate the right side is complete, the variable number that represents the left side variable is used to write a store statement that puts the content from zero into the variable on the left. If the ‘righthandside’ is simply a number or a character, only two intermediate code statements are written: one to store it in our working space, and one to store the working space into the ‘lefthandside’ variable.

If there is an expression on the right, the intermediate code that represents that mathematical expression is created from right to left, following the order of operations, and the result is always stored in the working space so it can easily be stored in the variable on the left in the end.

The code generation of conditional statements is handled when a ‘relop’ reduction is reached, but the CodeWriter class also helps when working with ‘if’ statements. If the expression simply contains ‘true’ then no conditional code is generated, but if the expression contains ‘false’, then a jump statement is written to jump over any code that is generated while passing through the ‘if ( false )’ statement. If there is a relational operator in the expression, a jump statement is written based on the operator in the expression that compares the two variables. The following table shows the relationship of operators and code generated:

* > Jump if less than or equal (JLE)
* >= Jump if less than (JLT)
* == Jump if not equal (JNE)
* <= Jump if greater than (JGT)
* < Jump if greater than or equal (JGE)
* <> Jump if equal (JEQ)

At the end of each jump statement, a temporary marker character is written, then when the ‘if’ statement is reduced all the way, that temporary marker is replaced with the current line number in the intermediate code file, this way, the entire center of the ‘if’ statement can be skipped over.

The only error messages that are thrown when generating code are I/O errors thrown by CodeWriter which are all caught by IOExceptions elsewhere.

## Outcome of unexpected termination

Should the program terminate unexpectedly, there is no need to worry. The source file used will not be corrupted or deleted, but the intermediate code file <FileName.asm> may be empty or incomplete and the executable file created with the -r option <FileName.asm.out> may also be incomplete or corrupt. Simply run the program again and it should end safely; if it crashes again, there is most likely a major problem with the source file, so correct the issue and run the program again.

# Appendix

## Program Examples

1. ConditionalInput

Source Text:

public static void CONDITIONAL (FIRST\_TERM, SECOND\_TERM)

{

var FIRST, SECOND, THIRD, FOURTH, FIFTH, SIXTH: int;

var LETTER, TRUE, FALSE, BLANK: char;

{

FIRST = 1;

SECOND = 2;

THIRD = 3;

FOURTH = 4;

FIFTH = 5;

SIXTH = 6;

LETTER = 'b';

TRUE = 'T';

FALSE = 'F';

BLANK = 10;//ASCII new line - this actually works

if (FIRST > SECOND)//this won't execute

{

put(FIRST)

};

if (FIRST >= SECOND)//this won't execute

{

put(SECOND)

};

if (FIRST == SECOND)//this won't execute

{

put(THIRD)

};

if(FIRST > TRUE)//this won’t execute

{

put(BLANK)

};

if (FIRST <= SECOND)

{

put(FOURTH);

if (FIRST <= SECOND)

{

put(BLANK)

}

};

if (FIRST < SECOND)

{

put(FIFTH)

};

if (FIRST <> SECOND)

{

put(SIXTH)

};

put(BLANK);

if(true)

{

put(TRUE)

};

if(false)

{

put(FALSE)//this should never execute

};

put(SECOND)

}

}

$

Program Output:

C:\Users\Matt\Dropbox\School\Compiler\CompilerDistributable>java -jar MDBCompiler.jar SampleFiles\ConditionalInput -r

Input accepted and compiled

Output intermediate code file: SampleFiles\ConditionalInput.asm

Intermediate Code mini-Assembler:

Running...

-----EOF (or blank line encountered)-----

mini-Intermediate Code Engine (mICE)

4

56

T2

\*\*\*ICE: Normal Program Termination

1. MathTester

Source Text:

public static void MATH (FIRST\_TERM)

{

var FIRST\_TERM, SECOND\_TERM, SUM: int;

var RESULT, BLANK: char;

{

FIRST\_TERM = 5;

SECOND\_TERM = 10;

BLANK = 10;//New line character in ASCII

SUM = FIRST\_TERM / 5 + 10;

put(SUM);

put(BLANK);

SUM = SECOND\_TERM;

put(SUM);

put(BLANK);

SUM = 30 \* 2 - 55;

put(SUM);

put(BLANK);

SUM = 3 % 2;

put(SUM);

put(BLANK);

put(FIRST\_TERM);

put(BLANK);

put(SECOND\_TERM)

}

}

$

Program Output:

C:\Users\Matt\Dropbox\School\Compiler\CompilerDistributable>java -jar MDBCompiler.jar SampleFiles\MathTester -r

Input accepted and compiled

Output intermediate code file: SampleFiles\MathTester.asm

Intermediate Code mini-Assembler:

Running...

-----EOF (or blank line encountered)-----

mini-Intermediate Code Engine (mICE)

11

10

5

1

5

10

\*\*\*ICE: Normal Program Termination

1. IOTester

Source Text:

public static void MATH (PARAMETER)

{

//Declare variables here

var INPUT, FIFTY: int;

var A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z, SPACE, COLON, LINE: char;

{

FIFTY = 50;

A='a'; B='b'; C='c'; D='d'; E='e'; F='f'; G='g'; H='h'; I='i'; J='j'; K='k'; L='l'; M='m';

N='n'; O='o'; P='p'; Q='q'; R='r'; S='s'; T='t'; U='u'; V='v'; W='w'; X='x'; Y='y'; Z='z';

SPACE = 32;//Space character in ASCII

COLON = 58;//Colon in ASCII

LINE = 10;//New line character in ASCII

put(LINE);

//enter a number - if less than 50, prints less than 50

put(E); put(N); put(T); put(E); put(R); put(SPACE); put(A); put(SPACE); put(N); put(U); put(M);

put(B); put(E); put(R); put(COLON);

get(INPUT);

if(INPUT < 50)

{

put(L); put(E); put(S); put(S); put(SPACE); put(T); put(H); put(A); put(N); put(SPACE);

put(FIFTY)

};

if(INPUT > 50)

{

put(G); put(R); put(E); put(A); put(T); put(E); put(R); put(SPACE); put(T); put(H); put(A); put(N); put(SPACE);

put(FIFTY)

};

if(INPUT == 50)

{

put(E); put(Q); put(U); put(A); put(L); put(SPACE); put(T); put(O); put(SPACE);

put(FIFTY)

};

put(LINE)

}

}

$

Program Output:

C:\Users\Matt\Dropbox\School\Compiler\CompilerDistributable>java -jar MDBCompiler.jar SampleFiles\IOTester -r

Input accepted and compiled

Output intermediate code file: SampleFiles\IOTester.asm

Intermediate Code mini-Assembler:

Running...

-----EOF (or blank line encountered)-----

mini-Intermediate Code Engine (mICE)

enter a number:49

less than 50

\*\*\*ICE: Normal Program Termination

C:\Users\Matt\Dropbox\School\Compiler\CompilerDistributable>java -jar MDBCompiler.jar SampleFiles\IOTester -r

Input accepted and compiled

Output intermediate code file: SampleFiles\IOTester.asm

Intermediate Code mini-Assembler:

Running...

-----EOF (or blank line encountered)-----

mini-Intermediate Code Engine (mICE)

enter a number:50

equal to 50

\*\*\*ICE: Normal Program Termination

C:\Users\Matt\Dropbox\School\Compiler\CompilerDistributable>java -jar MDBCompiler.jar SampleFiles\IOTester -r

Input accepted and compiled

Output intermediate code file: SampleFiles\IOTester.asm

Intermediate Code mini-Assembler:

Running...

-----EOF (or blank line encountered)-----

mini-Intermediate Code Engine (mICE)

enter a number:51

greater than 50

\*\*\*ICE: Normal Program Termination