Joel Christner (jec2160)
COMS-W4115 Spring 2009

Minimalistic Basic Compiler (MBC)

Project Final Report

Rev 1.0
5/14/2009

Table of Contents

Introduction

This document is the final project report for the Minimalistic Basic Compiler (MBC) project as completed by Joel Christner (UNI jec2160) for COMS-W4115, Spring 2009 semester (Professor Stephen Edwards).

MBC was designed to provide people with a compiler that would take simple BASIC-like source code and compile it into C source code. MBC does not implement all of the BASIC language, and restrictions exist on how the BASIC programs must be implemented, which will be discussed in this document.

Language Status and Open Defects

This section provides a status on the language and a list of open defects.

Language Status

MBC is largely implemented, but has a handful of open defects. The open defects were unable to be resolved by the time the project needed to be submitted.

Open Defects

The following are the open defects in MBC at time of submission.

ID	Description
1	Standalone mathematical expressions are not printed with a trailing semicolon in the resultant C code. Behavior not problematic for expressions in for, while, and if statements but is problematic for standalone mathematical expressions. Requires parser, AST, and backend modification.
2	No differentiation between assignment and comparison in use of 'equals' operator. Only a single-equals '=' is used, rather than a double-equals '=='. Requires parser, AST, and backend modification.
3	Incorrect expressions are being accepted by the parser (i.e. '20 = A'), and keywords are being allowed as identifiers. Numerical literals are being accepted as expressions. Requires parser, AST, and backend modification.
4	Errors not being thrown when problem encountered to allow the user to understand what line of code is causing a problem. Rather, a Parser error is generated. Backend modification required to correct to include try and raise statements to support exceptions. `Fatal error: exception Parsing.Parse_error`
5	REM statements are currently not functioning as expected. Requires parser, AST, and backend modification.

Language Tutorial

This section provides a high-level overview of MBC and how to use it.

Language Goals

The goal of MBC was to provide a simple tool that allowed simplistic BASIC programs to be compiled to C. With the introduction of DOS and eventually Windows operating systems, BASIC in its original form lost mindshare rapidly. Although dated, BASIC provides a solid programming foundation for those who are looking to learn a simple language that is not overwhelming or daunting in any way. MBC helps bridge the gap between learning BASIC and learning a more advanced language such as C.

Language Features

MBC supports integer data types. All BASIC statements must be contained within a single line and can not span lines. MBC supports certain BASIC program control, loop, and iteration statements, including 'END', 'FOR…NEXT', 'IF…THEN…ELSE' and 'WHILE'.

How to Use MBC

MBC is easy to use. Once compiled, simply run MBC from a command line and pass a BASIC source code file to it as input. For instance:

```
$ ./mbc < source.bas
```

The following examples show you how MBC works with various types of programs. Note that portions of text have been removed to keep the examples simple.

Mathematics Example

File basicmath.bas

```
a = 10
b = a - 2
c = b * 2
d = c / 4
print d
```

Compiler output

```
$ ./mbc < examples/math.bas
#include <stdlib.h>
#include <stdio.h>

int a;
int b;
int c;
int d;

void main() {

(a == 10)
(b == (a - 2))
(c == (b * 2))
(d == (c / 4))
printf(a);

}
```

Iteration (For Loop) Example

File forloop.bas

```
for a = 1 to 10 print a next
for b = 1 to 5 print b next
```

Compiler output

```
$ ./mbc < examples/forloop.bas
#include <stdlib.h>
#include <stdio.h>

int a;
int b;

void main() {
```

```
for ((a = 1), i < 10, i++) { printf(a); }
for ((b = 1), i < 5, i++) { printf(b); }
}
```

Iteration (While Loop) Example

File whileloop.bas

```
while a < 10 do a = a + 1 loop
while b < 100 do b = b + 20 loop
```

Compiler output

```
$ ./mbc < examples/whileloop.bas
#include <stdlib.h>
#include <stdio.h>

int a;
int b;

void main() {

while (a < 10) { (a = (a + 1))}
while (b < 100) { (b = (b + 20))}
}
```

Decision (If-Then-Else) Example

File ifthen.bas

```
a = 50
if a > 100 then print a else print 100
if a < 100 then print a else print 100
if a >= 50 then print a
```

Compiler output

```
$ ./mbc < examples/ifthen.bas
#include <stdlib.h>
#include <stdio.h>

int a;

void main() {

(a = 50)
if (a > 100) { printf(a); } else { printf(100); }
if (a < 100) { printf(a); } else { printf(100); }
if (a >= 50) { printf(a); }
}
```

Language Reference Manual

This section includes the language reference manual (LRM). The LRM presented here has been updated from the LRM that was provided earlier in the semester and reflects the current state of MBC as a language.

Language Overview

Minimalistic BASIC Compiler (MBC) provides a simple means of compiling small BASIC programs, and supports many of the commonly-used commands and features of the BASIC language. MBC will output C-compatible code which the user can then compile with a C compiler and execute.

Styles Used

This document uses three primary styles to visualize concepts. Standard document text as shown in this paragraph is in the Cambria font, 12 point. Text that shows an example of a line of MBC code is shown in `Courier New, 9 point`. Text contained within an example of a line of MBC code that should be considered a container for user-specified code is shown italicized in *`Courier New, 9 point`*., and will typically be encapsulated in braces (such as *`[insert code here]`*).

Lexical Conventions and Program Structure

This section defines the lexical conventions and general program structure used by the MBC language. MBC supports the ASCII character set only, and generally programs compiled with MBC are stored within a file. This section describes the general program structure and content requirements.

Program Termination

Programs will terminate on one of two conditions. The first being that there simply are no further lines of code to process, and the second being an explicit 'END' statement within the program.

Whitespace

White space characters are excluded during tokenization. This includes the space character ' ', carriage return '\r', tab '\t', and newline '\n' characters.

Program Structure

BASIC programs must have at most one statement per line. There are no numerical line identifiers.

Comments

MBC supports comments through the use of 'REM' within the program. The use of 'REM' indicates that any content from that point to the end of the line will be treated as a comment and not compiled. There are no multi-line comments.

```
REM <all characters through end of line are comments>

REM This is a comment
```

Tokens

There are several classes of tokens that can be used with MBC. These include identifiers, keywords, and numerical literals.

Identifiers

Identifiers are used to declare and reference stored information. Identifiers are a series of letters only and may be a mix of upper and lower case. Identifier names are case sensitive, that is, the identifier named `Variable` is not the same identifier as the identifier named `VARIABLE`. Identifiers do not need to be declared; identifiers are implicitly declared when used for the first time and are turned into global variables in the C program produced at the end. Only numerical identifiers are supported with MBC. Assignment to an identifier is done through the use of the assignment operator as denoted by the equals sign ('=') and using statements structured as follows:

```
[identifier] = [value]

A = 50
```

Should an identifier be used prior to a value being assigned, a zero value is used. The assignment operator (=) is also used within iteration statements and program control statements and in such cases is not providing assignment but is providing comparison. All identifiers in MBC are global; there is no concept of local identifiers in MBC.

Keywords

Keywords are those character strings that are reserved by MBC and cannot be used as names of identifiers, and keywords are not case sensitive. Keywords include:

REM	PRINT	FOR	TO
STEP	IF	THEN	ELSE
GOTO	GOSUB	RETURN	PRINT
END	DO	LOOP	WHILE

Each of these keywords will be explained throughout the course of this document.

Numerical Literals

Numerical literals are simply numbers (both integer as well as floating-point) that are used within a program and not assigned to an identifier. These must be on the right-side of an expression.

```
A = 5 + 10

A + 5
```

Miscellaneous

MBC does not support pointers or structures.

Operations

This section outlines the operations that are supported with MBC.

Multiplicative Operators

The multiplicative operator, denoted by asterisk for multiplication '*' or the slash '/' for division, provides the product of two integer literals or identifiers in the case of multiplication or the quotient of two integer literals or identifiers in the case of division. This can also be used in conjunction with print statements or in assignment. The multiplicative operators use the following syntax:

```
[product-identifier] = [id-or-val] * [id-or-val]

[quotient-identifier] = [id-or-val] / [id-or-val]

C = A * B

C = 5 * 20

C = A / 2

C = 100 / 5
```

Multiplicative operators are left associative. Multiplication holds higher precendence than division.

Additive Operators

The additive operator, denoted by the plus sign '+' for addition or the minus sign '-' for subtraction, provides the sum or difference of two integer literals or identifiers. The additive operators use the following syntax:

```
[sum-identifier] = [id-or-val] + [id-or-val]

[difference-identifier] = [id-or-val] - [id-or-val]

C = A + B

C = 5 + A

C = 5 - 2

C = D - 5
```

Additive operators are left associative. Addition holds higher precendence than subtraction, and both are at lower precedence than division.

Relational Operators

Relational operators define the relationship of two literals or identifiers. These are employed through the use of the greater than sign '>', the less than sign '<', the equals sign (=), or a combination of these, including:

- greater than or equal to: >=
- less than or equal to: <=
- not equal to: <> (either greater than or less than)

Assignment Operators

As mentioned above the equals sign (=) is used as an assignment operator, and places the value found on the right of the equals sign into the identifier named by the name on the left of the equals sign. The following syntax is used:

```
[identifier] = [value]
```

```
A = 50
```

```
A = B + 2
```

This operator is also used to determine equality or inequality in conditional statements that guide program flow or impact iterations. The exception to this rule is the 'FOR' statement which increments the value of an identifier by a certain amount (defined by 'STEP') upon each iteration.

Iteration Statements and Program Control

Iteration statements are used to execute statements in succession as long as the conditions associated with the iteration remain true. Iteration statements come in three forms: while loops, for loops, and if-then statements.

While Loops

The while loop executes the statements contained between the DO statement and the LOOP statement as the condition provided after 'while' is true. These statements follow this format:

```
WHILE [condition] DO [stmt] LOOP
```

```
WHILE A < 5 DO A = A + 1 LOOP
```

For Loops

A for-next loop executes a series of statements and automatically increments a numerical counter identifier by one after each iteration. Once the counter identifier reaches the amount specified in the value specified after 'TO', the statement block is executed for the final time. For-next loops have the following structure:

```
FOR [identifier] = startval TO endval [stmt] NEXT
```

```
FOR I = 1 TO 10 I = I + 1 NEXT
```

If-Then Statements

The if-then statement will evaluate a condition and then execute either the statement following 'THEN' if the condition is true, or the statement following ELSE if the condition is false. ELSE is optional, and if the condition following IF is not true, nothing is done. These statements have the following syntax:

```
IF [condition] THEN [stmt] ELSE [stmt]
```

```
IF [condition] THEN [stmt]

IF A = 5 THEN PRINT A ELSE PRINT 3

IF A = 5 THEN PRINT A
```

End Statements

The END statement will terminate the program when reached. Multiple END statements may exist within a program. An END statement is recommended but not required, and a program will terminate when no additional code exists to execute. The syntax for the END statement is simple:

```
END
```

Display Operations

The PRINT statement allows the user to display data on the standard output. When used by itself, PRINT will simply print a blank line. PRINT can be followed by a numerical literal or identifier. Syntax for the PRINT statement is:

```
PRINT

PRINT 5

PRINT A
```

Exceptions

MBC does not provide an exception-handling system.

Project Plan

This section provides details of the project plan in accordance with the sections requested on the class website and discussed in the course videos.

Processes

The following processes were used for planning, specification, development, and testing.

Item	Process
Planning	1. Identify high-level project concept 2. Determine high-level project components (toplevel, scanner, parser, AST) 3. Determine function of each high-level component and interaction amongst components
Specification	1. Examine each project component and determine internal function 2. Create pseudo-code to identify redundant code and create reusable functions
Development	1. Prepare source code skeletons for each high-level project component identified in planning 2. Expand pseudo-code defined in specification phase to implement functionality required
Testing	1. Identify test cases for individual components within the project 2. Identify test cases for the entire project 3. Create white-box and black-box test cases and stress boundary conditions

Programming Style

A number of commonly-used programming style elements were used in this project, including the following:

- Indentation and alignment – spaces from the left boundary in increments of two, and alignment under the previous function statement
- Comments – use them frequently to assist others using your code
- Warnings – do not ignore compiler warnings, eliminate them
- Parentheses – use them to eliminate ambiguity wherever necessary
- Spaces – use them between operators to make code more readable

Planned Project Timeline

The following shows the planned project timeline for MBC, and aligns each project task with the processes mentioned above. Please note that some dates are estimates as they were not originally logged.

Start Date	End Date	Description	Planning	Specification	Development	Testing
1/20/09	1/27/09	Brainstorming on project ideas, landed on MBC	X			
1/30/09	2/10/09	Development of proposal	X			
2/10/09	2/10/09	Submission of proposal	X			
2/17/09	2/24/09	Creation of skeleton program files	X		X	
2/24/09	3/10/09	Development of Language Resource Manual (LRM)	X	X		
3/10/09	3/10/09	Submission of LRM	X	X		
2/24/09	3/10/09	Design and development of scanner	X	X	X	
3/10/09	3/26/09	Design and development of parser	X	X	X	
3/17/09	3/31/09	Design and development of AST	X	X	X	
4/1/09	4/21/09	Design and development of toplevel including backend	X	X	X	
4/21/09	5/7/09	Testing				X
5/7/09	5/14/09	Creation of documentation for project final report				
5/14/09	5/14/09	Submission of final report and project tarball				

Roles and Responsibilities

I was the only contributor to this project. MBC was not planned, developed, or tested by a team. However, tactical issues were presented to subject matter experts for guidance where applicable.

Software Development Environment and Tools

The following describes the software development environment used during the project.

Item	Description
Operating System	Apple Macintosh OSX 10.5.6
Programming Language	Objective Caml 3.10.2 (including ocamlyacc, ocamllex)
Integrated Development Environment	vim and Apple XCode v3.1
Version Management	Filesystem folders
Documentation	Microsoft Office Word 2008
Presentation and Figures	Microsoft Office PowerPoint 2008
Books Used for Reference	Compilers: Principles, Techniques, & Tools (Aho, Lam, Sethi, Ullman) Practical OCaml (Smith)

	The Objective Caml Programming Language (Rentsch)
Helpful Resources	Caml Forums (http://caml.inria.fr/resources/forums.en.html) Caml Newsgroups Professor Edwards

Project Log

The following shows the actual project timeline for MBC, which can be compared and contrasted with the planned project timeline shown above. Please note that some dates are estimates as they were not originally logged.

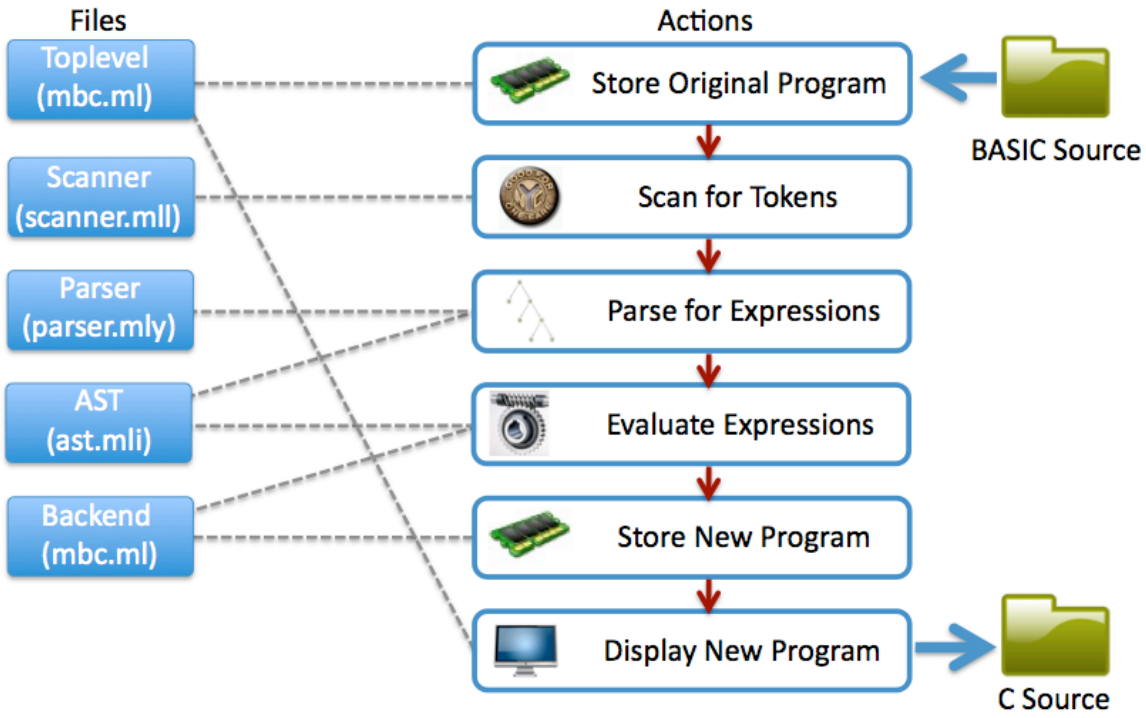
Start Date	End Date	Description	Planning	Specification	Development	Testing
1/20/09	1/27/09	Brainstorming on project ideas, landed on MBC	X			
1/30/09	2/10/09	Development of proposal	X			
2/10/09	2/10/09	Submission of proposal	X			
2/17/09	2/24/09	Creation of skeleton program files	X		X	
2/24/09	3/10/09	Development of Language Resource Manual (LRM)	X	X		
3/10/09	3/10/09	Submission of LRM	X	X		
3/24/09	4/9/09	Design and development of scanner	X	X	X	
3/31/09	4/21/09	Design and development of parser	X	X	X	
4/7/09	4/14/09	Design and development of AST	X	X	X	
4/21/09	5/14/09	Design and development of toplevel including backend	X	X	X	
5/9/09	5/14/09	Testing				X
5/12/09	5/14/09	Creation of documentation for project final report				
5/12/09	5/14/09	Last minute scramble to fix open issues			X	X
5/14/09	5/14/09	Submission of final report and project tarball				

Architectural Design

This section outlines the architectural design of MBC in accordance with the items requested on the course webpage.

Block Diagram of Major Components

A block diagram of MBC is shown below:



MBC consists of the following files, which perform the following functions:

Filename	Function
mbc.ml	This file provides both the toplevel (entry point into the program) as well as backend (processing of code).
scanner.mll	This file provides token definitions, which are used by the parser, by mapping regular expressions to token names. This file also handles comments as well as whitespace, thus turning file contents into a stream of tokens.
parser.mly	This file takes a stream of tokens and maps them to expressions. Additionally, associativity and precedence are defined to reduce conflicts associated with reduction and shifting.
ast.mli	This file provides the type definition and structure of expressions, which is used by the parser and by the backend.

Interfaces Between Components

The interfaces between the components of MBC are listed here:

Item 1	Item 2	Relationship and Interface
Toplevel	Original Source Program	Toplevel receives the original source program on stdin
Toplevel	Scanner	Toplevel sends lines from original source program to scanner to identify tokens
Toplevel	Parser	Tokens returned from scanner are then sent to the parser for identifying expressions using definitions set forth in the AST
Parser	AST	The AST defines the types and structures for the parser
Toplevel	Backend	The backend is part of the same file as toplevel (mbc.ml), evaluation of expressions occurs through the use of the function ‘eval’, which expects an expr as input (returned from the parser) and returns a string

Component Ownership

Each component in the system was owned and developed by Joel Christner.

Test Plan

This section outlines the test plan elements for MBC in accordance with the items requested on the course webpage.

Example Source Programs and Target Languages

The following table shows three different source language programs and the target language program for each.

Source Program	Target Program

`while a < 10 do a = a + 1 loop`
`while b < 100 do b = b + 20 loop`
	`#include <stdlib.h>`
	`#include <stdio.h>`
	
	`int a;`
	`int b;`
	
	`void main() {`
	
	`while (a < 10) { (a = (a + 1))}`
	`while (b < 100) { (b = (b + 20))}`
	`}`

`a = 10`
`b = 20`
`if a > b then a = a + 1 else b = b - 1`
`while a < b do a = a + 1 loop`
	`#include <stdlib.h>`
	`#include <stdio.h>`
	
	`int a;`
	`int b;`
	
	`void main() {`
	
	`(a = 10)`
	`(b = 20)`
	`if (a > b) { (a = (a + 1)) } else { (b = (b - 1)) }`
	`while (a < b) { (a = (a + 1))}`
	`}`

`a = 10`
`b = a + 10`
`for c = 1 to 20 a = a + 1 next`
`if a > c then print a else print c`
`print b`
	`#include <stdlib.h>`
	`#include <stdio.h>`
	
	`int a;`
	`int b;`
	`int c;`
	
	`void main() {`
	
	`(a = 10)`
	`(b = (a + 10))`
	`for ((c = 1), i < 20, i++) { (a = (a + 1)) }`
	`if (a > c) { printf(a); } else {`
	`printf(c); }`
	`printf(b);`
	`}`

Translator Test Suites

A series of BASIC programs known to either be valid or invalid were created and stored in the testcases/ directory. Files containing programs known to be valid end in –good.bas, whereas files containing programs known to be problematic end in –bad.bas. These BASIC programs cover virtually every aspect of MBC, including addition subtraction, assignment, for loops, identifiers, if-then statements, literals, multiplication, division, print, relationships, comments, and while loops.

These tests can be executed individually from the command line using:

```
$ ./mbc < testcases/<filename.bas>
```

Additionally, a shell script has been created, stored in the root mbc directory (called test.sh) which will execute each of these tests in sequence:

```
$ ./test.sh
```

If you wish, you can pipe the output of the tests to a file using standard notation:

```
$ ./test.sh > testresults
```

Additionally, these tests can be executed through the 'make' command as follows:

```
$ make test
```

Test Case Reasoning

These tests were chosen specifically to ensure that MBC operated correctly under normal conditions and also under abnormal conditions.

Automation

No automation was used in the testing of MBC. All testing was performed manually.

Component Ownership

Each component in the system was owned by Joel Christner.

Lessons Learned

This section covers the lessons learned through the course of the class and in particular this project, and then provides advice for future teams or individuals that are taking this course.

Most Important Learnings

The most important learnings through this project and throughout the course include

- Most of us find it easy to take a language and a compiler for granted. Understanding the intricacies of language construction and how compilers work under the hood is helpful in developing sound, efficient, and performant software
- Common compilers can be broken into a small number of key components including scanning (turning streams of characters into tokens), syntax trees (organization and structure of expressions and other types), parsing (identifying those expressions and other types), and the backend (rules defining what to do when expressions and other types are encountered)
- Objective Caml is a powerful functional language that also includes imperative features, and is language that provides streamlined compiler development

- Objective Caml is particularly powerful – in my humble opinion – on the front-end of a compiler, specifically scanning and parsing. Getting the backend right in Objective Caml is difficult, and I would have preferred to have used an imperative language such as C for this task

Advice for Future Teams

The following are advisory items for future teams that are building projects for COMS-W4115 and for future students that are taking the class.

- For those that do not have experience programming in a functional language (I didn't), start practicing with Objective Caml early. Study and rebuild the calculator example, mapping out the way in which pieces work together. Study the microc example as well, and also look at past projects
- Start your project early, do not procrastinate. There is no worse feeling than being at the end of the eleventh hour with a massive bug when you should already be done with testing and have your documentation nearly complete
- Have laser-focus on your project. Follow the KISS philosophy (keep it simple stupid). You won't have time to get to your 'stretch' items. It's better to do a small number of things right than to do a large number of things terribly wrong
- Allocate sufficient time weekly to study and to practice using Objective Caml. You don't want to learn this language in the final half of the class
- Print a hard copy of all slides that are related to the project and to Objective Caml. Keep them in a binder and record your notes there, using tabs as bookmarks for quick access
- Start with the scanner first. It is easiest. You'll find that you work with it less and less as your project moves forward
- Develop the parser and AST second. These must be done together, as the AST defines the types used by the parser. Be sure associativity and precedence are defined correctly
- Develop a generic backend and toplevel first, just to make sure your statements are tokenized and parsed correctly into expressions. Think 'printf'

Appendix

Complete Code Listing

Below is the output of each of the source code files. None of the files generated by the Objective Caml compiler or other tools are included. Each of these files was authored by Joel Christner.

scanner.mll

```
(* scanner.mll :: mbc scanner :: jec2160 *)
{ open Parser }

let whitespace = [' ' '\t' '\r' '\n']

rule token = parse
  whitespace { token lexbuf }
| "REM"      { comment lexbuf }
| '('        { LPAREN }
| ')'        { RPAREN }
| ':'        { COLON }
| ','        { COMMA }
| '+'        { PLUS }
| '-'        { MINUS }
| '*'        { TIMES }
| '/'        { DIVIDE }
| '='        { EQ }
| '<'        { LT }
```

```
| '>'         { GT }
| "<>"        { NEQ }
| "<="        { LEQ }
| ">="        { GEQ }
| "if"        { IF }
| "then"      { THEN }
| "else"      { ELSE }
| "for"       { FOR }
| "to"        { TO }
| "step"      { STEP }
| "next"      { NEXT }
| "while"     { WHILE }
| "do"        { DO }
| "loop"      { LOOP }
| "goto"      { GOTO }
| "gosub"     { GOSUB }
| "return"    { RETURN }
| "end"       { END }
| "print"     { PRINT }
| "input"     { INPUT }

(* literals and identifiers              *)
| ['0'-'9']+ as lit              { LITERAL(int_of_string lit) }
| ['a'-'z' 'A'-'Z']+ as lxm      { ID(lxm) }

(* extra                                 *)
| eof        { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

(* comment handler                       *)
and comment = parse
  '\n'       { token lexbuf }
| _          { comment lexbuf }
```

parser.mly

```
/* parser.ml :: mbc parser :: jec2160 */
%{ open Ast %}

/* tokens */
%token LPAREN RPAREN COLON COMMA
%token PLUS MINUS TIMES DIVIDE
%token EQ NEQ LT GT LEQ GEQ
%token IF THEN ELSE FOR TO STEP NEXT
%token WHILE DO LOOP GOTO
%token GOSUB RETURN END PRINT INPUT
%token EOF
%token <int> LITERAL
%token <string> ID

/* associativity and precedence */
%nonassoc ELSE
%left EQ NEQ
%left GT GEQ LT LEQ
%left PLUS MINUS
%left TIMES DIVIDE

/* change start to stmt after implemented */
%start expr
%type < Ast.expr> expr

%%

expr:
```

```
  | ID                              { Id($1) }
  | expr   PLUS   expr              { Binop($1, Add, $3, $3) }
  | expr   MINUS  expr              { Binop($1, Sub, $3, $3) }
  | expr   TIMES  expr              { Binop($1, Mul, $3, $3) }
  | expr   DIVIDE expr              { Binop($1, Div, $3, $3) }
  | expr   EQ     expr              { Binop($1, Equal, $3, $3) }
  | expr   NEQ    expr              { Binop($1, Notequal, $3, $3) }
  | expr   LT     expr                 { Binop($1, Less, $3, $3) }
  | expr   GT     expr              { Binop($1, Greater, $3, $3) }
  | expr   LEQ    expr              { Binop($1, Leq, $3, $3) }
  | expr   GEQ    expr              { Binop($1, Geq, $3, $3) }
  | LPAREN expr RPAREN              { $2 }
  | LITERAL                         { Lit($1) }
  | IF expr THEN expr ELSE expr { Binop($2, Ifthenelse, $4, $6) }
  | IF expr THEN expr           { Binop($2, Ifthen, $4, $4) }
  | WHILE expr DO expr LOOP     { Binop($2, Whileop, $4, $4) }
  | FOR expr TO expr expr NEXT  { Binop($2, Forop, $4, $5) }
  | PRINT expr                  { Binop($2, Printop, $2, $2) }
  | EOF                                          { Endop }
  | END                                          { Endop }
```

ast.mli

```
type operator = Add | Sub | Mul | Div | Equal | Notequal | Less | Greater | Leq
| Geq
  | Ifthenelse | Ifthen | Whileop | Forop | Printop

type expr =
  | Binop of expr * operator * expr * expr
  | Id of string
  | Lit of int
  | Endop
```

mbc.ml

```
(* mbc.ml :: mbc toplevel :: jec2160 *)
open Ast

let debuglevel = 0  (* 1=debug, 0=no debug *)

(* add variables to a list, no concern over duplication *)
let add_text variablelist text =
  variablelist := text::!variablelist

(* add de-duplicated variables to a list - use 'add_text_nodup' *)
let rec add_to_list variablelist text =
  match variablelist with
  | h::t -> if h = text then variablelist else h::add_to_list t text
  | [] -> [text]

let add_text_nodup variablelist text =
  variablelist := add_to_list !variablelist text

(* top of program *)
let programheader1 =   ref ["#include <stdlib.h>"; "#include <stdio.h>\n"]

(* middle of program, printed after variable list *)
let programheader2 =   ref ["\nvoid main() {\n"]

(* end of program *)
let programfooter1 =   ref ["}\n"]

(* output program body *)
let (programcontents : (string list) ref) = ref []

(* original program body *)
let (originalprogramcontents : (string list) ref) = ref []
```

```
(* list of variables that were defined, all are global *)
let (varlist : (string list) ref) = ref []

(* print all variable definitions *)
let printallvariables varlist = List.iter
  (fun n -> print_string "int ";
            print_string n;
             print_string ";\n";;) varlist

(* print list forward *)
let printlistfwd varlist =
  List.iter (fun n -> print_string n; print_string "\n") !varlist

(* print list backward *)
let printlistback varlist =
  List.iter (fun n -> print_string n; print_string "\n") (List.rev(!varlist))


let rec eval = function
  | Lit(x) -> string_of_int x              (* number *)

  | Id(x) ->                               (* variable *)
      if (debuglevel == 1) then
         print_string ("variable name " ^ x ^ "\n") else print_string "";
       add_text_nodup varlist x;
       x

  | Endop ->                               (* end of program *)
      if (debuglevel == 1) then
        print_string "endop\n" else print_string "";
       ""

  | Binop(e1, op, e2, e3) ->               (* binary operators *)
      let v1 = eval e1 and v2 = eval e2 and v3 = eval e3 in
      match op with

       | Add ->                             (* addition *)
          if (debuglevel == 1) then
            print_string "addition\n" else print_string "";
          "(" ^ v1 ^ " + " ^ v2 ^ ")"

      | Sub ->                             (* subtraction *)
          if (debuglevel == 1) then
            print_string "subtraction\n" else print_string "";
          "(" ^ v1 ^ " - " ^ v2 ^ ")"

      | Mul ->                             (* multiplication *)
          if (debuglevel == 1) then
            print_string "multiplication\n" else print_string "";
          "(" ^ v1 ^ " * " ^ v2 ^ ")"

      | Div ->                             (* division *)
          if (debuglevel == 1) then
            print_string "division\n" else print_string "";
          "(" ^ v1 ^ " / " ^ v2 ^ ")"

       | Equal ->                           (* equal *)
          if (debuglevel == 1) then
            print_string "equal\n" else print_string "";
          "(" ^ v1 ^ " = " ^ v2 ^ ")"

       | Notequal ->                        (* not equal *)
          if (debuglevel == 1) then
            print_string "notequal\n" else print_string "";
          "(" ^ v1 ^ " != " ^ v2 ^ ")"

       | Less ->                            (* less than *)
          if (debuglevel == 1) then
            print_string "less\n" else print_string "";
          "(" ^ v1 ^ " < " ^ v2 ^ ")"
```

```
       | Greater ->                          (* greater than *)
         if (debuglevel == 1) then
           print_string "greater\n" else print_string "";
         "(" ^ v1 ^ " > " ^ v2 ^ ")"

       | Leq ->                              (* less than or equal *)
         if (debuglevel == 1) then
           print_string "lessequal\n" else print_string "";
         "(" ^ v1 ^ " <= " ^ v2 ^ ")"

       | Geq ->                              (* greater than or equal *)
         if (debuglevel == 1) then
           print_string "greaterequal\n" else print_string "";
         "(" ^ v1 ^ " >= " ^ v2 ^ ")"

       | Ifthenelse ->                       (* if then else *)
         if (debuglevel == 1) then
           print_string "ifthenelse\n" else print_string "";
         "if " ^ v1 ^ " { " ^ v2 ^ " } else { " ^ v3 ^ " }"

       | Ifthen ->                           (* if then no else *)
         if (debuglevel == 1) then
           print_string "ifthen\n" else print_string "";
         "if " ^ v1 ^ " { " ^ v2 ^ " }"

       | Whileop ->                          (* while do loop *)
         if (debuglevel == 1) then
           print_string "whileop\n" else print_string "";
         "while " ^ v1 ^ " { " ^ v2 ^ "}"

       | Forop ->                            (* for loop *)
         if (debuglevel == 1) then
           print_string "forop\n" else print_string "";
         "for (" ^ v1 ^ ", i < " ^ v2 ^ ", i++) { " ^ v3 ^ " }"

      | Printop ->                          (* printop *)
         if (debuglevel == 1) then
           print_string "printop\n" else print_string "";
         "printf(" ^ v1 ^ ");"

let _ =
  (* welcome message *)
  print_endline "\n\nminimalistic basic compiler :: version 1.0 :: joel christner";
  print_endline "project for coms-w4115 :: columbia university :: prof edwards\n";
  print_endline "original program contents";
  print_endline "------------------------------------------------------------------------
-------";

  (* gather original program contents *)
  let rec gatherprogram () =
    try
      let line = input_line stdin in
         (* print_endline line;      <-- remove for debugging *)
       add_text originalprogramcontents line;
         gatherprogram ()
    with End_of_file ->
      printlistback originalprogramcontents;       (* display originalprogramcontents *)
      ()
  in
    gatherprogram ();

  (* end of gathering program contents *)
  print_endline "\n\n----------------------------------------------------------------------
-----------";
  print_endline "end of input file, processing...";
  print_endline "------------------------------------------------------------------------
-------";
  (* now we have the entire program stored in originalprogramcontents *)

  (* evaluate expressions from each line in originalprogramcontents, store result in
programcontents *)
```

```
  let programparser =
    List.iter (fun programline ->
                (* print_string ": ";          *)
                (* print_string programline;   *)
                      (* print_string "\n";          *)
                  let lexbuf = Lexing.from_string programline in
                       let expr = Parser.expr Scanner.token lexbuf in
                         let result = eval expr in
                           print_endline result;
                               (* now we have each line evaluated *)
                               (* need to add it to programcontents *)
                               add_text programcontents result
                ) (List.rev(!originalprogramcontents))
  in
    programparser;

  (* final program stored in programcontents - need to display *)
  print_endline "\n\n-------------------------------------------------------------------------
-----------";
  print_endline "output file...";
  print_endline "-------------------------------------------------------------------------
-------";

  (* print programheader1 *)
  printlistfwd programheader1;

  (* print list of variables *)
  printallvariables !varlist;

  (* print programheader2 *)
  printlistfwd programheader2;

  (* print programcontents *)
  printlistback programcontents;

  (* print programfooter1 *)
  printlistfwd programfooter1;
  print_endline "\n\n-------------------------------------------------------------------------
-----------";
  print_endline "finished...";
  print_endline "-------------------------------------------------------------------------
-------";
```

makefile

```
OBJS = parser.cmo scanner.cmo mbc.cmo

TESTS = \
addsub-bad \
addsub-good \
assign-bad \
assign-good \
for-bad \
for-good \
forloops-bad \
forloops-good \
identifiers-bad \
identifiers-good \
ifthen-bad \
ifthen-good \
literals \
multdiv-bad \
multdiv-good \
print \
relation-bad \
relation-good \
rem \
while-bad \
```

```
while-good

EXAMPLES = \
exampleprogram1 \
exampleprogram2 \
forloop \
ifthen \
math \
whileloop

TARFILES = Makefile scanner.mll parser.mly ast.mli mbc.ml tests.sh \
           $(TESTS:%=testcases/%.bas) \
          $(EXAMPLES:%=examples/%.bas)

mbc : $(OBJS)
     ocamlc -o mbc $(OBJS)

.PHONY : test
test : mbc test.sh
     ./test.sh

scanner.ml : scanner.mll
     ocamllex scanner.mll

parser.ml parser.mli : parser.mly
     ocamlyacc parser.mly

%.cmo : %.ml
     ocamlc -c $<

%.cmi : %.mli
     ocamlc -c $<

mbc.tar.gz : $(TARFILES)
     cd .. && tar zcf mbc/mbc.tar.gz $(TARFILES:%=mbc/%)

.PHONY : clean
clean :
     rm -f mbc parser.ml parser.mli scanner.ml *.cmo *.cmi

# Generated by ocamldep *.ml *.mli
mbc.cmo: scanner.cmo parser.cmi ast.cmi
mbc.cmx: scanner.cmx parser.cmx ast.cmi
parser.cmo: ast.cmi parser.cmi
parser.cmx: ast.cmi parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmi
```

End of Document

This concludes the final report for the MBC project. I thoroughly enjoyed this class and this project, and only wish I had a couple more days to wrap up the loose ends. Thanks to Professor Edwards for managing such a challenging and rewarding course.