Halit Özsoy
2016400141

04.11.2017
CMPE 230 – HOMEWORK 1

REPORT / DOCUMENTATION FOR HOMEWORK 1

Problem:

Develop a program that converts bitc instructions into an .asm code for a86
"./bitc [input_file].bc" should create a file "[input_file].asm"

How I Solved The Problem

I decided to solve the problem by writing a cpp program.

Then, started the project by creating todo functions, to reduce the problem into subproblems.
1 – Parse bitc instructions
2 – Convert them to assembly

1a – find print operations
1b – find variables
1c – iteratively parse each operation (so that the program would work very fast)

2a – print general a86 start & end code parts
2b – print assembly print functions
2c – print variables
2d – print operations

For 2d, using a reverse polish notation (postfix) for operations was the solution I used.
For other parts of the 2nd problem, I studied a86, and learned to create each part myself, than wrote cpp functions for those.

For 1a, I searched for '=' in each line, and decided the line is to be printed if non is found.

For 1b, I inserted each variable I parsed into a set to extract unique variables.

For 1c, I decided to use another subproblem instead of 1c. That would be 1d.

1d – parse each line of operation by a right recursive way and by reducing them to expressions, terms and factors.

Challenges I Faced During the Project

I received an error, "definition conflicts with forward reference," which took an hour for me to solve. I learned that although it is written in the end, I had to write down the size of variables (b/w) near operations when using only them in an operation. (for example, for pushing them to stack)
That was the way I managed to fix this bug.

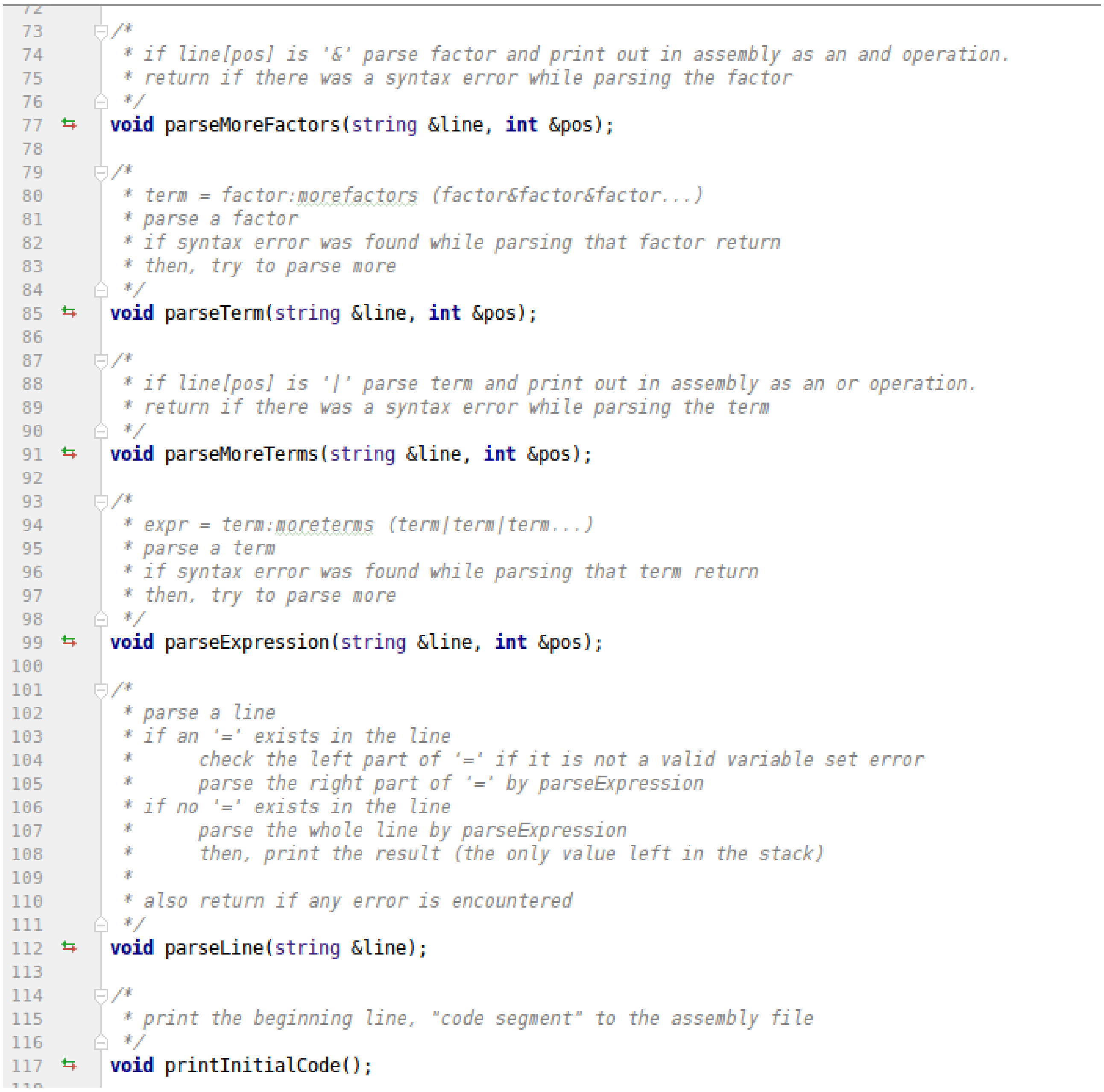
What I Learned

I have gained some practice on assembly a86, and learned how to code for a86 by doing. I managed to write print functions myself. Printing numbers in hexadecimal form was a little challenge for me which made me learn more about jumps, comparisons, functions and specific registers in assembly.

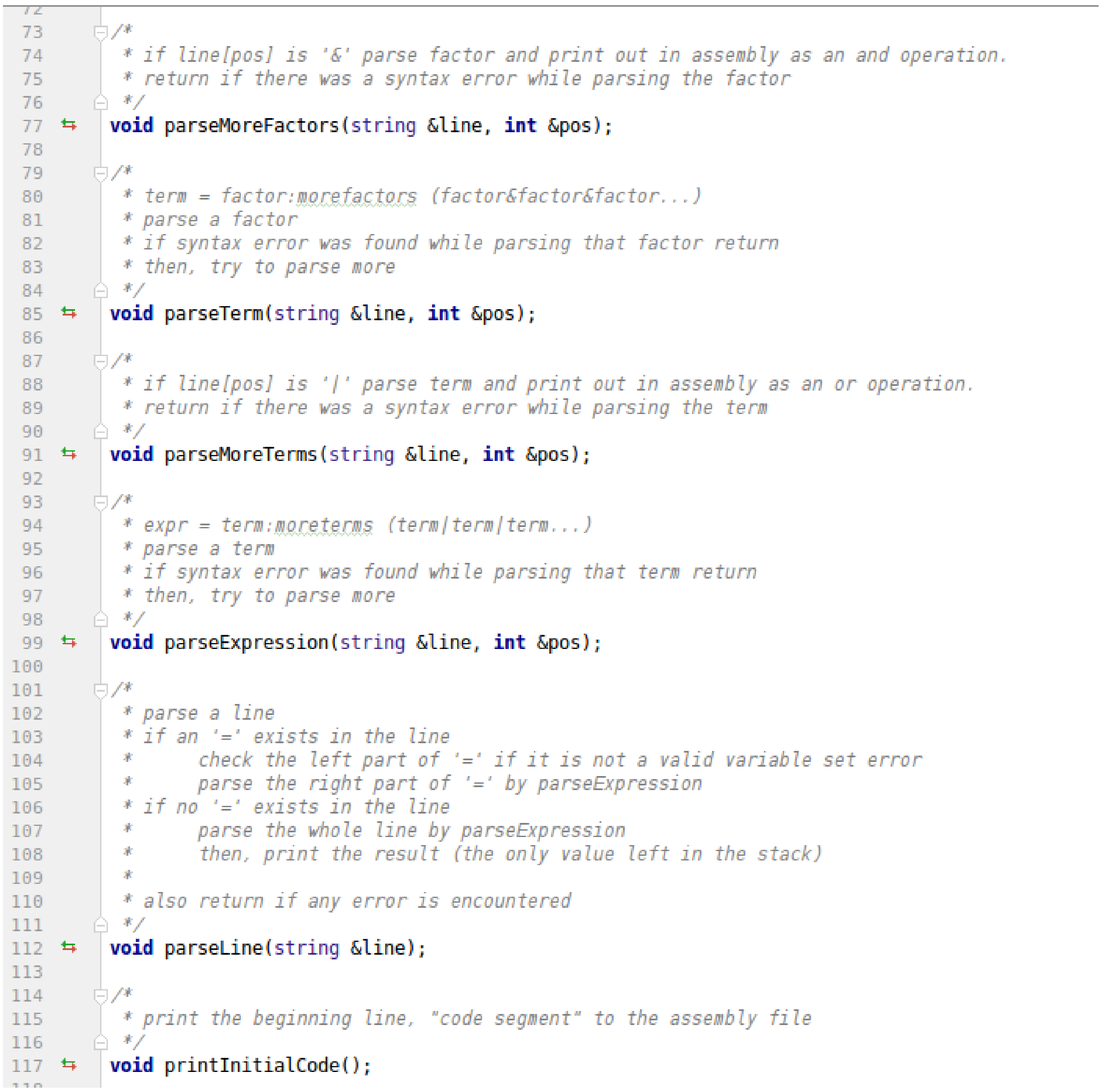
Halit Özsoy
2016400141

04.11.2017
CMPE 230 – HOMEWORK 1

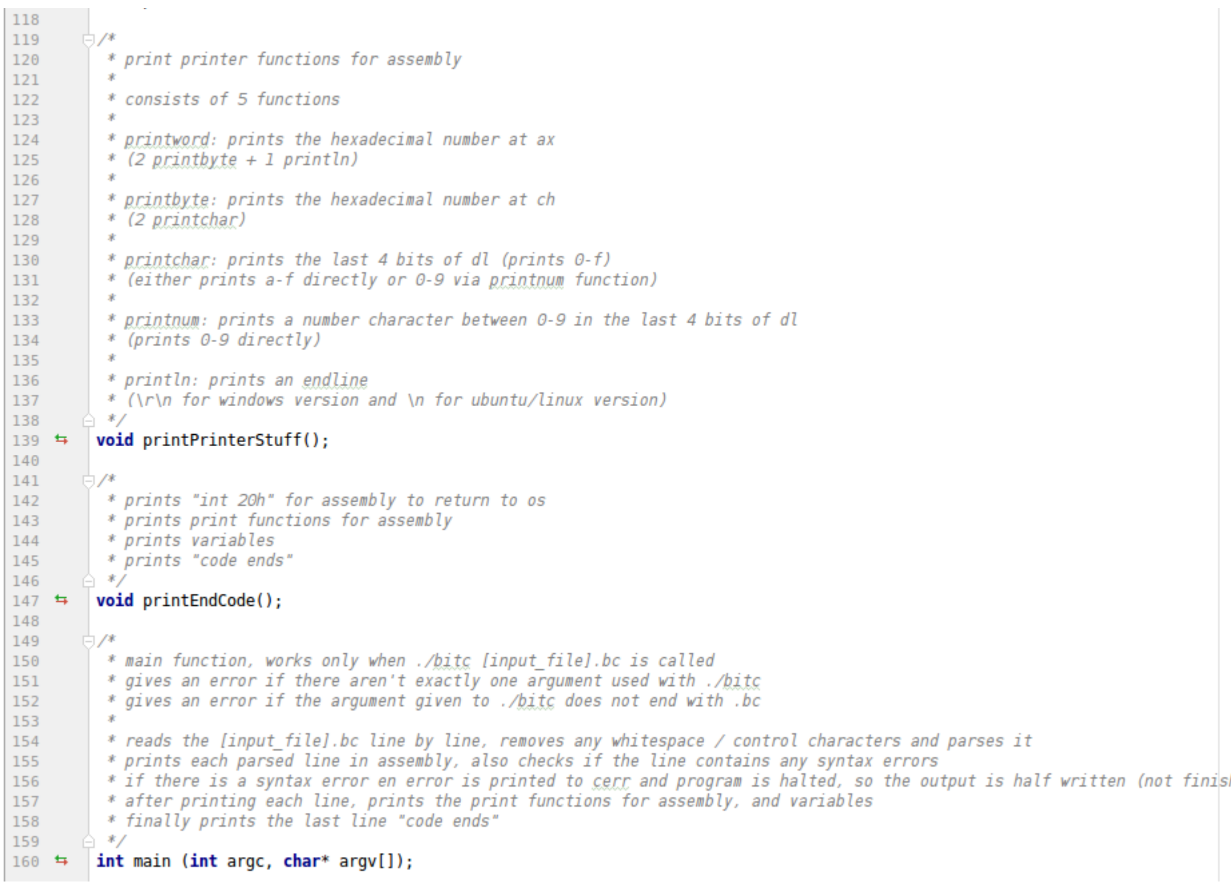
Documentation



```
/*
 * if line[pos] is '&' parse factor and print out in assembly as an and operation.
 * return if there was a syntax error while parsing the factor
 */
void parseMoreFactors(string &line, int &pos);

/*
 * term = factor:morefactors (factor&factor&factor...)
 * parse a factor
 * if syntax error was found while parsing that factor return
 * then, try to parse more
 */
void parseTerm(string &line, int &pos);

/*
 * if line[pos] is '|' parse term and print out in assembly as an or operation.
 * return if there was a syntax error while parsing the term
 */
void parseMoreTerms(string &line, int &pos);

/*
 * expr = term:moreterms (term|term|term...)
 * parse a term
 * if syntax error was found while parsing that term return
 * then, try to parse more
 */
void parseExpression(string &line, int &pos);

/*
 * parse a line
 * if an '=' exists in the line
 *       check the left part of '=' if it is not a valid variable set error
 *       parse the right part of '=' by parseExpression
 * if no '=' exists in the line
 *       parse the whole line by parseExpression
 *       then, print the result (the only value left in the stack)
 *
 * also return if any error is encountered
 */
void parseLine(string &line);

/*
 * print the beginning line, "code segment" to the assembly file
 */
void printInitialCode();
```



```
/*
 * if line[pos] is '&' parse factor and print out in assembly as an and operation.
 * return if there was a syntax error while parsing the factor
 */
void parseMoreFactors(string &line, int &pos);

/*
 * term = factor:morefactors (factor&factor&factor...)
 * parse a factor
 * if syntax error was found while parsing that factor return
 * then, try to parse more
 */
void parseTerm(string &line, int &pos);

/*
 * if line[pos] is '|' parse term and print out in assembly as an or operation.
 * return if there was a syntax error while parsing the term
 */
void parseMoreTerms(string &line, int &pos);

/*
 * expr = term:moreterms (term|term|term...)
 * parse a term
 * if syntax error was found while parsing that term return
 * then, try to parse more
 */
void parseExpression(string &line, int &pos);

/*
 * parse a line
 * if an '=' exists in the line
 *      check the left part of '=' if it is not a valid variable set error
 *      parse the right part of '=' by parseExpression
 * if no '=' exists in the line
 *      parse the whole line by parseExpression
 *      then, print the result (the only value left in the stack)
 *
 * also return if any error is encountered
 */
void parseLine(string &line);

/*
 * print the beginning line, "code segment" to the assembly file
 */
void printInitialCode();
```



```

/*
 * print printer functions for assembly
 *
 * consists of 5 functions
 *
 * printword: prints the hexadecimal number at ax
 * (2 printbyte + 1 println)
 *
 * printbyte: prints the hexadecimal number at ch
 * (2 printchar)
 *
 * printchar: prints the last 4 bits of dl (prints 0-f)
 * (either prints a-f directly or 0-9 via printnum function)
 *
 * printnum: prints a number character between 0-9 in the last 4 bits of dl
 * (prints 0-9 directly)
 *
 * println: prints an endline
 * (\r\n for windows version and \n for ubuntu/linux version)
 */
void printPrinterStuff();

/*
 * prints "int 20h" for assembly to return to os
 * prints print functions for assembly
 * prints variables
 * prints "code ends"
 */
void printEndCode();

/*
 * main function, works only when ./bitc [input_file].bc is called
 * gives an error if there aren't exactly one argument used with ./bitc
 * gives an error if the argument given to ./bitc does not end with .bc
 *
 * reads the [input_file].bc line by line, removes any whitespace / control characters and parses it
 * prints each parsed line in assembly, also checks if the line contains any syntax errors
 * if there is a syntax error en error is printed to cerr and program is halted, so the output is half written (not finished)
 * after printing each line, prints the print functions for assembly, and variables
 * finally prints the last line "code ends"
 */
int main (int argc, char* argv[]);
```