Jose Franco Baquera

April 30, 2018

CS – 370

Lab 9 - Create NASM code from your AST

Here is the .c code that was used for my demonstration with Dr. Cooper. This code checks for reading, writing, while, if-else, arrays, assignments, and function calls. (NOTE: This file can be found in the same zip folder and is called finalTest.c)

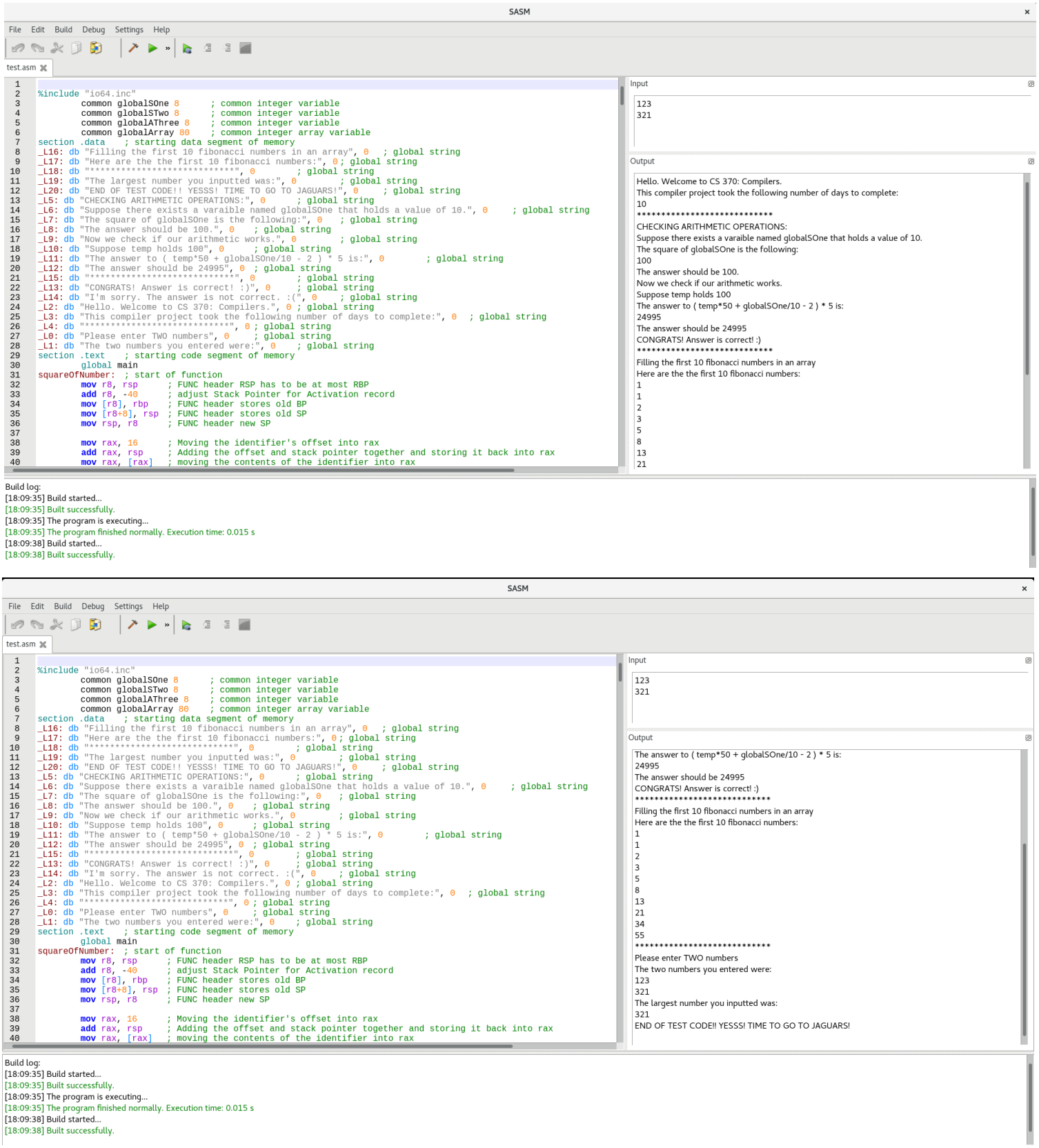
```
int globalSOne;
int globalSTwo;
int globalAThree;
int globalArray[10];
int squareOfNumber( int x ) {

   x = x * x;
   return x;

}

int readANumbersFromUser( void ) {

   write "Please enter TWO numbers";
   read globalSOne;
   read globalSTwo;
   write "The two numbers you entered were:";
   write globalSOne;
   write globalSTwo;

}

int fibonacci ( int number ) {

   if( number <= 0 ) {
      return 0;
   }
   if( number == 1){
      return 1;
   }
   if( number == 2 ) {
      return 1;
   }

   return fibonacci( number - 1 ) + fibonacci( number - 2 );
```

```
}

int maxOfTwoNumbers( int x, int y ) {

    if( x > y ) {
      return x;
    }

    if ( x < y ) {
      return y;
    }

    if( x == y ) {
      return x;
    }

}

void printGreetingMessage( void ) {

   int numberOfDays;
   numberOfDays = 10;
   write "Hello. Welcome to CS 370: Compilers.";
   write "This compiler project took the following number of days to complete:";
   write numberOfDays;
   write "*****************************";

}

void checkArithmeticOperations( void ) {

   int temp;
   globalSOne = 10;
   write "CHECKING ARITHMETIC OPERATIONS:";
   write "Suppose there exists a varaible named globalSOne that holds a value of 10.";
   write "The square of globalSOne is the following:";
   temp = squareOfNumber( globalSOne );
   write temp;
   write "The answer should be 100.";
   write "Now we check if our arithmetic works.";
   write "Suppose temp holds 100";
   write "The answer to ( temp*50 + globalSOne/10 - 2 ) * 5 is:";
   write ( temp*50 + globalSOne/10 - 2 ) * 5;
   write "The answer should be 24995";
   if( ( ( temp*50 + globalSOne/10 - 2 ) * 5 ) == 24995 ) {
     write "CONGRATS! Answer is correct! :)";
   }
   else {
     write "I'm sorry. The answer is not correct. :(";
```

```
  }

  write "*****************************";

}

void main ( void ) {

  int counter;
  int temp;
  counter = 0;
  printGreetingMessage( );
  checkArithmeticOperations( );
  write "Filling the first 10 fibonacci numbers in an array";
  write "Here are the the first 10 fibonacci numbers:";

  while( counter < 10 ) {
    globalArray[counter]= fibonacci(counter + 1);
    counter = counter + 1;
  }

  counter = 0;

  while( counter < 10 ) {
    write globalArray[counter];
    counter = counter +1;
  }

  write "*****************************";

  readANumbersFromUser( );
  temp = maxOfTwoNumbers( globalSOne, globalSTwo );
  write "The largest number you inputted was:";
  write temp;
  write "END OF TEST CODE!! YESSS! TIME TO GO TO JAGUARS!";


}
```

(Screenshots are found on next page!)

Here is a screenshot of my NASM environment:



The above code produced the following sasm assembly code (NOTE: This can also be found in the same zip folder and it is called test.sasm):

```
%include "io64.inc"
common globalSOne 8 ; common integer variable
common globalSTwo 8 ; common integer variable
common globalAThree 8 ; common integer variable
common globalArray 80 ; common integer array variable
section .data ; starting data segment of memory
_L16: db "Filling the first 10 fibonacci numbers in an array", 0 ; global string
_L17: db "Here are the the first 10 fibonacci numbers:", 0 ; global string
_L18: db "*****************************", 0 ; global string
_L19: db "The largest number you inputted was:", 0 ; global string
_L20: db "END OF TEST CODE!! YESSS! TIME TO GO TO JAGUARS!", 0 ; global string
_L5: db "CHECKING ARITHMETIC OPERATIONS:", 0 ; global string
_L6: db "Suppose there exists a varaible named globalSOne that holds a value of 10.", 0 ; global string
_L7: db "The square of globalSOne is the following:", 0 ; global string
_L8: db "The answer should be 100.", 0 ; global string
_L9: db "Now we check if our arithmetic works.", 0 ; global string
_L10: db "Suppose temp holds 100", 0 ; global string
_L11: db "The answer to ( temp*50 + globalSOne/10 - 2 ) * 5 is:", 0 ; global string
_L12: db "The answer should be 24995", 0 ; global string
_L15: db "*****************************", 0 ; global string
_L13: db "CONGRATS! Answer is correct! :)", 0 ; global string
_L14: db "I'm sorry. The answer is not correct. :(", 0 ; global string
_L2: db "Hello. Welcome to CS 370: Compilers.", 0 ; global string
_L3: db "This compiler project took the following number of days to complete:", 0 ; global string
_L4: db "*****************************", 0 ; global string
_L0: db "Please enter TWO numbers", 0 ; global string
_L1: db "The two numbers you entered were:", 0 ; global string
section .text ; starting code segment of memory
global main
squareOfNumber: ; start of function
mov r8, rsp ; FUNC header RSP has to be at most RBP
add r8, -40 ; adjust Stack Pointer for Activation record
mov [r8], rbp ; FUNC header stores old BP
mov [r8+8], rsp ; FUNC header stores old SP
mov rsp, r8 ; FUNC header new SP


mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
```

```
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +24], rax ; storing the left hand side into the offset
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rbx, [rax] ; moving the contents of the identifier into rbx
mov rax, [rsp + 24] ; fetching the left hand side
imul rax, rbx ; multiply expression
mov [rsp + 24], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 24] ; moving to register rax the contents of the expression
mov [rsp + 32], rax ; storing the final value of EXPRSTMT into its offset
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rbx, [32 + rsp] ; Storing the final value of the expression into a temporary register
mov [rax], rbx ; Store the value into the identifier's memory address
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; move the contents of the identifier's the memory address into rax


mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
ret ; return to the caller


mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
ret ; return to the caller
readANumbersFromUser: ; start of function
mov r8, rsp ; FUNC header RSP has to be at most RBP
add r8, -16 ; adjust Stack Pointer for Activation record
mov [r8], rbp ; FUNC header stores old BP
mov [r8+8], rsp ; FUNC header stores old SP
mov rsp, r8 ; FUNC header new SP


PRINT_STRING _L0 ; print a string
NEWLINE ;standard Write a NEWLINE
mov rax, globalSOne ; storing the memory address of the global variable into rax
GET_DEC 8, [rax] ; READ in an integer
mov rax, globalSTwo ; storing the memory address of the global variable into rax
GET_DEC 8, [rax] ; READ in an integer
PRINT_STRING _L1 ; print a string
```

```
NEWLINE ;standard Write a NEWLINE
mov rax, globalSOne ; storing the memory address of the global variable into rax
mov rsi, [rax] ; move the contents of the identifier's the memory address into rsi
PRINT_DEC 8, rsi ;standard Write a value
NEWLINE ; standard Write a NEWLINE
mov rax, globalSTwo ; storing the memory address of the global variable into rax
mov rsi, [rax] ; move the contents of the identifier's the memory address into rsi
PRINT_DEC 8, rsi ;standard Write a value
NEWLINE ; standard Write a NEWLINE


mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
ret ; return to the caller
fibonacci: ; start of function
mov r8, rsp ; FUNC header RSP has to be at most RBP
add r8, -88 ; adjust Stack Pointer for Activation record
mov [r8], rbp ; FUNC header stores old BP
mov [r8+8], rsp ; FUNC header stores old SP
mov rsp, r8 ; FUNC header new SP


mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +24], rax ; storing the left hand side into the offset
mov rbx, 0 ; moving a number into register rbx
mov rax, [rsp + 24] ; fetching the left hand side
cmp rax, rbx ; EXPR lessthanequal
setle al ; sets the last bit in the last byte of the rax register
mov rbx, 1 ; set rbx to one to filter rax
and rax, rbx ; filter rax by using the 'and' assembly instruction
mov [rsp + 24], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 24] ; moving to register rax the contents of the expression
cmp rax, 0 ; comparing register rax to 0. If equal, that means that the expression is false
je _L21 ; IF branch out


mov rax, 0 ; moving a number into register rax


mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
```

```
ret ; return to the caller
jmp _L22 ; jumping out of the 'if' body

_L21: ; label that will allow us to skip the if's body if the expression is false

_L22: ; Label to exit the else part
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +32], rax ; storing the left hand side into the offset
mov rbx, 1 ; moving a number into register rbx
mov rax, [rsp + 32] ; fetching the left hand side
cmp rax, rbx ; EXPR equal
sete al ; sets the last bit in the last byte of the rax register
mov rbx, 1 ; set rbx to one to filter rax
and rax, rbx ; filter rax by using the 'and' assembly instruction
mov [rsp + 32], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 32] ; moving to register rax the contents of the expression
cmp rax, 0 ; comparing register rax to 0. If equal, that means that the expression is false
je _L23 ; IF branch out

mov rax, 1 ; moving a number into register rax

mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
ret ; return to the caller
jmp _L24 ; jumping out of the 'if' body

_L23: ; label that will allow us to skip the if's body if the expression is false

_L24: ; Label to exit the else part
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +40], rax ; storing the left hand side into the offset
mov rbx, 2 ; moving a number into register rbx
mov rax, [rsp + 40] ; fetching the left hand side
cmp rax, rbx ; EXPR equal
```

```
sete al ; sets the last bit in the last byte of the rax register
mov rbx, 1 ; set rbx to one to filter rax
and rax, rbx ; filter rax by using the 'and' assembly instruction
mov [rsp + 40], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 40] ; moving to register rax the contents of the expression
cmp rax, 0 ; comparing register rax to 0. If equal, that means that the expression is false
je _L25 ; IF branch out


mov rax, 1 ; moving a number into register rax


mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
ret ; return to the caller
jmp _L26 ; jumping out of the 'if' body


_L25: ; label that will allow us to skip the if's body if the expression is false


_L26: ; Label to exit the else part
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +48], rax ; storing the left hand side into the offset
mov rbx, 1 ; moving a number into register rbx
mov rax, [rsp + 48] ; fetching the left hand side
sub rax, rbx ; subtraction expression
mov [rsp + 48], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 48] ; moving to register rax the contents of the expression
mov [rsp + 56], rax ; Storing the evaluated argument into ARGLIST's offset
mov rcx, rsp ; Store the current stack pointer into a trivial register (i.e. rcx in this case)
sub rcx, 96 ; Subtract the size of the function (PLUS 1) in order to determine were the function is
located on the stack
mov rax, [rsp + 56] ; Move the contents of the argument into a temporary register
mov [rcx + 16], rax ; 'Copy' the contents of rax into the parameter's memory address
call fibonacci ; calling a function HELLO
mov [rsp +80], rax ; storing the left hand side into the offset
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +64], rax ; storing the left hand side into the offset
```

```
mov rbx, 2 ; moving a number into register rbx
mov rax, [rsp + 64] ; fetching the left hand side
sub rax, rbx ; subtraction expression
mov [rsp + 64], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 64] ; moving to register rax the contents of the expression
mov [rsp + 72], rax ; Storing the evaluated argument into ARGLIST's offset
mov rcx, rsp ; Store the current stack pointer into a trivial register (i.e. rcx in this case)
sub rcx, 96 ; Subtract the size of the function (PLUS 1) in order to determine were the function is
located on the stack
mov rax, [rsp + 72] ; Move the contents of the argument into a temporary register
mov [rcx + 16], rax ; 'Copy' the contents of rax into the parameter's memory address
call fibonacci ; calling a function
mov rbx, rax ;
mov rax, [rsp + 80] ; fetching the left hand side
add rax, rbx ; addition expression
mov [rsp + 80], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 80] ; moving to register rax the contents of the expression


mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
ret ; return to the caller


mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
ret ; return to the caller
maxOfTwoNumbers: ; start of function
mov r8, rsp ; FUNC header RSP has to be at most RBP
add r8, -56 ; adjust Stack Pointer for Activation record
mov [r8], rbp ; FUNC header stores old BP
mov [r8+8], rsp ; FUNC header stores old SP
mov rsp, r8 ; FUNC header new SP


mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +32], rax ; storing the left hand side into the offset
mov rax, 24 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rbx, [rax] ; moving the contents of the identifier into rbx
```

```
mov rax, [rsp + 32] ; fetching the left hand side
cmp rax, rbx ; EXPR greaterthan
setg al ; sets the last bit in the last byte of the rax register
mov rbx, 1 ; set rbx to one to filter rax
and rax, rbx ; filter rax by using the 'and' assembly instruction
mov [rsp + 32], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 32] ; moving to register rax the contents of the expression
cmp rax, 0 ; comparing register rax to 0. If equal, that means that the expression is false
je _L27 ; IF branch out

mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; move the contents of the identifier's the memory address into rax

mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
ret ; return to the caller
jmp _L28 ; jumping out of the 'if' body

_L27: ; label that will allow us to skip the if's body if the expression is false

_L28: ; Label to exit the else part
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +40], rax ; storing the left hand side into the offset
mov rax, 24 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rbx, [rax] ; moving the contents of the identifier into rbx
mov rax, [rsp + 40] ; fetching the left hand side
cmp rax, rbx ; EXPR lessthan
setl al ; sets the last bit in the last byte of the rax register
mov rbx, 1 ; set rbx to one to filter rax
and rax, rbx ; filter rax by using the 'and' assembly instruction
mov [rsp + 40], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 40] ; moving to register rax the contents of the expression
cmp rax, 0 ; comparing register rax to 0. If equal, that means that the expression is false
je _L29 ; IF branch out
```

```
mov rax, 24 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; move the contents of the identifier's the memory address into rax

mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
ret ; return to the caller
jmp _L30 ; jumping out of the 'if' body

_L29: ; label that will allow us to skip the if's body if the expression is false

_L30: ; Label to exit the else part
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +48], rax ; storing the left hand side into the offset
mov rax, 24 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rbx, [rax] ; moving the contents of the identifier into rbx
mov rax, [rsp + 48] ; fetching the left hand side
cmp rax, rbx ; EXPR equal
sete al ; sets the last bit in the last byte of the rax register
mov rbx, 1 ; set rbx to one to filter rax
and rax, rbx ; filter rax by using the 'and' assembly instruction
mov [rsp + 48], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 48] ; moving to register rax the contents of the expression
cmp rax, 0 ; comparing register rax to 0. If equal, that means that the expression is false
je _L31 ; IF branch out

mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; move the contents of the identifier's the memory address into rax

mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
ret ; return to the caller
jmp _L32 ; jumping out of the 'if' body
```

```
_L31: ; label that will allow us to skip the if's body if the expression is false

_L32: ; Label to exit the else part

mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
ret ; return to the caller
printGreetingMessage: ; start of function
mov r8, rsp ; FUNC header RSP has to be at most RBP
add r8, -32 ; adjust Stack Pointer for Activation record
mov [r8], rbp ; FUNC header stores old BP
mov [r8+8], rsp ; FUNC header stores old SP
mov rsp, r8 ; FUNC header new SP

mov rax, 10 ; moving a number into register rax
mov [rsp + 24], rax ; storing the final value of EXPRSTMT into its offset
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rbx, [24 + rsp] ; Storing the final value of the expression into a temporary register
mov [rax], rbx ; Store the value into the identifier's memory address
PRINT_STRING _L2 ; print a string
NEWLINE ;standard Write a NEWLINE
PRINT_STRING _L3 ; print a string
NEWLINE ;standard Write a NEWLINE
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rsi, [rax] ; move the contents of the identifier's the memory address into rsi
PRINT_DEC 8, rsi ;standard Write a value
NEWLINE ; standard Write a NEWLINE
PRINT_STRING _L4 ; print a string
NEWLINE ;standard Write a NEWLINE

mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
ret ; return to the caller
checkArithmeticOperations: ; start of function
mov r8, rsp ; FUNC header RSP has to be at most RBP
add r8, -136 ; adjust Stack Pointer for Activation record
mov [r8], rbp ; FUNC header stores old BP
```

```
mov [r8+8], rsp ; FUNC header stores old SP
mov rsp, r8 ; FUNC header new SP


mov rax, 10 ; moving a number into register rax
mov [rsp + 24], rax ; storing the final value of EXPRSTMT into its offset
mov rax, globalSOne ; storing the memory address of the global variable into rax
mov rbx, [24 + rsp] ; Storing the final value of the expression into a temporary register
mov [rax], rbx ; Store the value into the identifier's memory address
PRINT_STRING _L5 ; print a string
NEWLINE ;standard Write a NEWLINE
PRINT_STRING _L6 ; print a string
NEWLINE ;standard Write a NEWLINE
PRINT_STRING _L7 ; print a string
NEWLINE ;standard Write a NEWLINE
mov rax, globalSOne ; storing the memory address of the global variable into rax
mov rax, [rax] ; move the contents of the identifier's the memory address into rax
mov [rsp + 32], rax ; Storing the evaluated argument into ARGLIST's offset
mov rcx, rsp ; Store the current stack pointer into a trivial register (i.e. rcx in this case)
sub rcx, 48 ; Subtract the size of the function (PLUS 1) in order to determine were the function is
located on the stack
mov rax, [rsp + 32] ; Move the contents of the argument into a temporary register
mov [rcx + 16], rax ; 'Copy' the contents of rax into the parameter's memory address
call squareOfNumber ; calling a function
mov [rsp + 40], rax ; storing the final value of EXPRSTMT into its offset
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rbx, [40 + rsp] ; Storing the final value of the expression into a temporary register
mov [rax], rbx ; Store the value into the identifier's memory address
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rsi, [rax] ; move the contents of the identifier's the memory address into rsi
PRINT_DEC 8, rsi ;standard Write a value
NEWLINE ; standard Write a NEWLINE
PRINT_STRING _L8 ; print a string
NEWLINE ;standard Write a NEWLINE
PRINT_STRING _L9 ; print a string
NEWLINE ;standard Write a NEWLINE
PRINT_STRING _L10 ; print a string
NEWLINE ;standard Write a NEWLINE
PRINT_STRING _L11 ; print a string
```

```
NEWLINE ;standard Write a NEWLINE
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +48], rax ; storing the left hand side into the offset
mov rbx, 50 ; moving a number into register rbx
mov rax, [rsp + 48] ; fetching the left hand side
imul rax, rbx ; multiply expression
mov [rsp + 48], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 48] ; moving the contents of the expression into rax
mov [rsp +64], rax ; storing the left hand side into the offset
mov rax, globalSOne ; storing the memory address of the global variable into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +56], rax ; storing the left hand side into the offset
mov rbx, 10 ; moving a number into register rbx
mov rax, [rsp + 56] ; fetching the left hand side
xor rdx, rdx ; xor rdx to clear the register
idiv rbx ; dividing rax by rbx
mov [rsp + 56], rax ; Storing the final answer of the expression into its offset
mov rbx, [rsp + 56] ; moving the contents of the expression into rbx
mov rax, [rsp + 64] ; fetching the left hand side
add rax, rbx ; addition expression
mov [rsp + 64], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 64] ; moving the contents of the expression into rax
mov [rsp +72], rax ; storing the left hand side into the offset
mov rbx, 2 ; moving a number into register rbx
mov rax, [rsp + 72] ; fetching the left hand side
sub rax, rbx ; subtraction expression
mov [rsp + 72], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 72] ; moving the contents of the expression into rax
mov [rsp +80], rax ; storing the left hand side into the offset
mov rbx, 5 ; moving a number into register rbx
mov rax, [rsp + 80] ; fetching the left hand side
imul rax, rbx ; multiply expression
mov [rsp + 80], rax ; Storing the final answer of the expression into its offset
mov rsi, [rsp + 80] ; moving to register rsi the contents of the expression
PRINT_DEC 8, rsi ;standard Write a value
NEWLINE ; standard Write a NEWLINE
PRINT_STRING _L12 ; print a string
NEWLINE ;standard Write a NEWLINE
mov rax, 16 ; Moving the identifier's offset into rax
```

```
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +88], rax ; storing the left hand side into the offset
mov rbx, 50 ; moving a number into register rbx
mov rax, [rsp + 88] ; fetching the left hand side
imul rax, rbx ; multiply expression
mov [rsp + 88], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 88] ; moving the contents of the expression into rax
mov [rsp +104], rax ; storing the left hand side into the offset
mov rax, globalSOne ; storing the memory address of the global variable into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +96], rax ; storing the left hand side into the offset
mov rbx, 10 ; moving a number into register rbx
mov rax, [rsp + 96] ; fetching the left hand side
xor rdx, rdx ; xor rdx to clear the register
idiv rbx ; dividing rax by rbx
mov [rsp + 96], rax ; Storing the final answer of the expression into its offset
mov rbx, [rsp + 96] ; moving the contents of the expression into rbx
mov rax, [rsp + 104] ; fetching the left hand side
add rax, rbx ; addition expression
mov [rsp + 104], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 104] ; moving the contents of the expression into rax
mov [rsp +112], rax ; storing the left hand side into the offset
mov rbx, 2 ; moving a number into register rbx
mov rax, [rsp + 112] ; fetching the left hand side
sub rax, rbx ; subtraction expression
mov [rsp + 112], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 112] ; moving the contents of the expression into rax
mov [rsp +120], rax ; storing the left hand side into the offset
mov rbx, 5 ; moving a number into register rbx
mov rax, [rsp + 120] ; fetching the left hand side
imul rax, rbx ; multiply expression
mov [rsp + 120], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 120] ; moving the contents of the expression into rax
mov [rsp +128], rax ; storing the left hand side into the offset
mov rbx, 24995 ; moving a number into register rbx
mov rax, [rsp + 128] ; fetching the left hand side
cmp rax, rbx ; EXPR equal
sete al ; sets the last bit in the last byte of the rax register
mov rbx, 1 ; set rbx to one to filter rax
and rax, rbx ; filter rax by using the 'and' assembly instruction
```

```
mov [rsp + 128], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 128] ; moving to register rax the contents of the expression
cmp rax, 0 ; comparing register rax to 0. If equal, that means that the expression is false
je _L33 ; IF branch out

PRINT_STRING _L13 ; print a string
NEWLINE ;standard Write a NEWLINE
jmp _L34 ; jumping out of the 'if' body

_L33: ; label that will allow us to skip the if's body if the expression is false
PRINT_STRING _L14 ; print a string
NEWLINE ;standard Write a NEWLINE

_L34: ; Label to exit the else part
PRINT_STRING _L15 ; print a string
NEWLINE ;standard Write a NEWLINE

mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
ret ; return to the caller
main: ; start of function
mov rbp, rsp ; SPECIAL RSP to RSB for MAIN only
mov r8, rsp ; FUNC header RSP has to be at most RBP
add r8, -112 ; adjust Stack Pointer for Activation record
mov [r8], rbp ; FUNC header stores old BP
mov [r8+8], rsp ; FUNC header stores old SP
mov rsp, r8 ; FUNC header new SP

mov rax, 0 ; moving a number into register rax
mov [rsp + 32], rax ; storing the final value of EXPRSTMT into its offset
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rbx, [32 + rsp] ; Storing the final value of the expression into a temporary register
mov [rax], rbx ; Store the value into the identifier's memory address
mov rcx, rsp ; Store the current stack pointer into a trivial register (i.e. rcx in this case)
sub rcx, 40 ; Subtract the size of the function (PLUS 1) in order to determine were the function is
located on the stack
call printGreetingMessage ; calling a function
mov [rsp + 40], rax ; storing the final value of EXPRSTMT into its offset
```

```
mov rcx, rsp ; Store the current stack pointer into a trivial register (i.e. rcx in this case)
sub rcx, 144 ; Subtract the size of the function (PLUS 1) in order to determine were the function is located on the stack
call checkArithmeticOperations ; calling a function
mov [rsp + 48], rax ; storing the final value of EXPRSTMT into its offset
PRINT_STRING _L16 ; print a string
NEWLINE ;standard Write a NEWLINE
PRINT_STRING _L17 ; print a string
NEWLINE ;standard Write a NEWLINE


_L35: ; start of while
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +56], rax ; storing the left hand side into the offset
mov rbx, 10 ; moving a number into register rbx
mov rax, [rsp + 56] ; fetching the left hand side
cmp rax, rbx ; EXPR lessthan
setl al ; sets the last bit in the last byte of the rax register
mov rbx, 1 ; set rbx to one to filter rax
and rax, rbx ; filter rax by using the 'and' assembly instruction
mov [rsp + 56], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 56] ; moving to register rax the contents of the expression
cmp rax, 0 ; comparing register rax to 0. If equal, that means that the expression is false
je _L36 ; while branch out


mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +64], rax ; storing the left hand side into the offset
mov rbx, 1 ; moving a number into register rbx
mov rax, [rsp + 64] ; fetching the left hand side
add rax, rbx ; addition expression
mov [rsp + 64], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 64] ; moving to register rax the contents of the expression
mov [rsp + 72], rax ; Storing the evaluated argument into ARGLIST's offset
mov rcx, rsp ; Store the current stack pointer into a trivial register (i.e. rcx in this case)
sub rcx, 96 ; Subtract the size of the function (PLUS 1) in order to determine were the function is located on the stack
mov rax, [rsp + 72] ; Move the contents of the argument into a temporary register
```

```
mov [rcx + 16], rax ; 'Copy' the contents of rax into the parameter's memory address
call fibonacci ; calling a function
mov [rsp + 80], rax ; storing the final value of EXPRSTMT into its offset
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; move the contents of the identifier's the memory address into rax
shl rax, 3 ; shift rax left by 3 (i.e. this will simulate multiplying by 8)
mov rbx, rax ; Storing the internal offset of an array into rbx
mov rax, globalArray ; storing the memory address of the global variable into rax
add rax, rbx ; adding the internal offset to the regular offset if we encounter an array
mov rbx, [80 + rsp] ; Storing the final value of the expression into a temporary register
mov [rax], rbx ; Store the value into the identifier's memory address
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +88], rax ; storing the left hand side into the offset
mov rbx, 1 ; moving a number into register rbx
mov rax, [rsp + 88] ; fetching the left hand side
add rax, rbx ; addition expression
mov [rsp + 88], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 88] ; moving to register rax the contents of the expression
mov [rsp + 96], rax ; storing the final value of EXPRSTMT into its offset
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rbx, [96 + rsp] ; Storing the final value of the expression into a temporary register
mov [rax], rbx ; Store the value into the identifier's memory address
jmp _L35 ; jump back to check expression

_L36: ; end of while
mov rax, 0 ; moving a number into register rax
mov [rsp + 64], rax ; storing the final value of EXPRSTMT into its offset
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rbx, [64 + rsp] ; Storing the final value of the expression into a temporary register
mov [rax], rbx ; Store the value into the identifier's memory address

_L37: ; start of while
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
```

```
mov [rsp +72], rax ; storing the left hand side into the offset
mov rbx, 10 ; moving a number into register rbx
mov rax, [rsp + 72] ; fetching the left hand side
cmp rax, rbx ; EXPR lessthan
setl al ; sets the last bit in the last byte of the rax register
mov rbx, 1 ; set rbx to one to filter rax
and rax, rbx ; filter rax by using the 'and' assembly instruction
mov [rsp + 72], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 72] ; moving to register rax the contents of the expression
cmp rax, 0 ; comparing register rax to 0. If equal, that means that the expression is false
je _L38 ; while branch out

mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; move the contents of the identifier's the memory address into rax
shl rax, 3 ; shift rax left by 3 (i.e. this will simulate multiplying by 8)
mov rbx, rax ; Storing the internal offset of an array into rbx
mov rax, globalArray ; storing the memory address of the global variable into rax
add rax, rbx ; adding the internal offset to the regular offset if we encounter an array
mov rsi, [rax] ; move the contents of the identifier's the memory address into rsi
PRINT_DEC 8, rsi ;standard Write a value
NEWLINE ; standard Write a NEWLINE
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rax, [rax] ; moving the contents of the identifier into rax
mov [rsp +80], rax ; storing the left hand side into the offset
mov rbx, 1 ; moving a number into register rbx
mov rax, [rsp + 80] ; fetching the left hand side
add rax, rbx ; addition expression
mov [rsp + 80], rax ; Storing the final answer of the expression into its offset
mov rax, [rsp + 80] ; moving to register rax the contents of the expression
mov [rsp + 88], rax ; storing the final value of EXPRSTMT into its offset
mov rax, 16 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rbx, [88 + rsp] ; Storing the final value of the expression into a temporary register
mov [rax], rbx ; Store the value into the identifier's memory address
jmp _L37 ; jump back to check expression

_L38: ; end of while
PRINT_STRING _L18 ; print a string
```

```
NEWLINE ;standard Write a NEWLINE
mov rcx, rsp ; Store the current stack pointer into a trivial register (i.e. rcx in this case)
sub rcx, 24 ; Subtract the size of the function (PLUS 1) in order to determine were the function is located on the stack
call readANumbersFromUser ; calling a function
mov [rsp + 80], rax ; storing the final value of EXPRSTMT into its offset
mov rax, globalSOne ; storing the memory address of the global variable into rax
mov rax, [rax] ; move the contents of the identifier's the memory address into rax
mov [rsp + 96], rax ; Storing the evaluated argument into ARGLIST's offset
mov rax, globalSTwo ; storing the memory address of the global variable into rax
mov rax, [rax] ; move the contents of the identifier's the memory address into rax
mov [rsp + 88], rax ; Storing the evaluated argument into ARGLIST's offset
mov rcx, rsp ; Store the current stack pointer into a trivial register (i.e. rcx in this case)
sub rcx, 64 ; Subtract the size of the function (PLUS 1) in order to determine were the function is located on the stack
mov rax, [rsp + 96] ; Move the contents of the argument into a temporary register
mov [rcx + 16], rax ; 'Copy' the contents of rax into the parameter's memory address
mov rax, [rsp + 88] ; Move the contents of the argument into a temporary register
mov [rcx + 24], rax ; 'Copy' the contents of rax into the parameter's memory address
call maxOfTwoNumbers ; calling a function
mov [rsp + 104], rax ; storing the final value of EXPRSTMT into its offset
mov rax, 24 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rbx, [104 + rsp] ; Storing the final value of the expression into a temporary register
mov [rax], rbx ; Store the value into the identifier's memory address
PRINT_STRING _L19 ; print a string
NEWLINE ;standard Write a NEWLINE
mov rax, 24 ; Moving the identifier's offset into rax
add rax, rsp ; Adding the offset and stack pointer together and storing it back into rax
mov rsi, [rax] ; move the contents of the identifier's the memory address into rsi
PRINT_DEC 8, rsi ;standard Write a value
NEWLINE ; standard Write a NEWLINE
PRINT_STRING _L20 ; print a string
NEWLINE ;standard Write a NEWLINE


mov rbp, [rsp] ; FUNC end restore old BP
mov rsp, [rsp+8] ; FUNC end restore old SP
mov rsp, rbp ; stack and BP need to be same on exit for main
ret ; return to the caller
```

Here is a copy of my EMITAST.c

```
/*

   Author's Name: Jose Franco Baquera

   Project Name:  Lab 9 - Create NASM code from your AST

   Program Name: EMITAST.c

   Date: April 25, 2018

   Class: CS 370 - Shaun Cooper

   Lab Number: 9

   Purpose/Description: The purpose of this C file is to produce SASM/NASAM
   assembly code into a file which name is provided by the user. NOTE: The input
   file MUST have a main function, otherwise when it is runned in SASM compiling errors
   will be produced. There are a total of 10 functions in this file. The "beginning"
   function that will be called from YACC is emitAST and its parameters are the AST node
   (in this case it will be the root node) and a FILE object.

*/

// Include several header files that will be used to
// produce the assembly-equivalent code.
#include "EMITAST.h"
#include "symtable.h"
#include "Lab6AST.h"

// Define a variable that will be used to multiply
// the correct number of words needed. NOTE: A word is
// is 8 BYTES.
#define DW 8

// Define a char variable that will keep track of the current function we
// are processing. This will basically allow us to distinguish non-main functions
// vs the main function.
char * functionName;

// Precondition: The parameter node points to
// an AST node type IDENT.
// Postcondition: Register RAX will hold the
// memory address of the identifier.
void emitIdentifier( FILE *fp, ASTnode *p ) {

   if( mydebug )
      printf( "Entering emitIdentifier function.\n" );
```

```
  // First check if the identifier is an array. If it is, caculate its internal
  // offset. NOTE: We will temporary store the "value" of the internal offset into RAX,
  // then into RBX.
  if( p->stmt0 != NULL ) {

    if( mydebug )
      printf( "Found an array identifier. Calculating internal offset.\n" );

    switch( p->stmt0->type ) {
      case NUMBER:
        // If the type of stmt0 is NUMBER, store that number into rax.
        fprintf( fp, "\tmov rax, %d\t; moving a number into register rax\n", p->stmt0->value);
        break;
      case IDENT:
        // If the type of stmt0 is IDENT, we need its memory address. Find the memory address by
calling emitIdentifier
        // and storing the contents of that memory address into RAX.
        emitIdentifier(fp, p->stmt0);
        fprintf( fp, "\tmov rax, [rax]\t; move the contents of the identifier's the memory address into
rax\n");
        break;
      case EXPR:
        // If the type of stmt0 is EXPR, we need to call emitExpression. After this function ends, the
result will be stored
        // in the expression's offset. Find the contents of this offset and store them into register RAX.
        emitExpression( fp, p->stmt0);
        fprintf( fp, "\tmov rax, [rsp + %d]\t; moving to register rax the contents of the expression\n", p-
>stmt0->symbol->offset * DW );
        break;
      case CALL:
        // If the type of stmt0 is CALL, call emitFunction. We then call the function. After the function
returns,
        // RAX will already have the value we are looking for.
        emitFunction( fp, p->stmt0 );
        fprintf( fp, "\tcall %s\t; calling a function\n", p->stmt0->name );
        break;
      default: fprintf(fp, "\t; Unknown target in emitIdentifier.\n");

    } // end switch

    // Multiply the internal offset by 8.
    fprintf( fp, "\tshl rax, 3\t; shift rax left by 3 (i.e. this will simulate multiplying by 8)\n" );

    // Store the temporary internal offset into RBX.
    fprintf( fp, "\tmov rbx, rax\t; Storing the internal offset of an array into rbx\n" );

  } // end if.
```

```
   // If the identifier is a global variable, we can use the identifier's name
   // to store its memory address into RAX.
   if ( p->symbol->level == 0 )
      fprintf( fp, "\tmov rax, %s\t; storing the memory address of the global variable into rax\n", p->name
);

   // If the identifier is not a global variable, we must use its offset. NOTE: We must multiply
   // the offset by DW, which in this case is 8.
   else {
      fprintf( fp, "\tmov rax, %d\t; Moving the identifier's offset into rax\n", p->symbol->offset * DW);
      fprintf( fp, "\tadd rax, rsp\t; Adding the offset and stack pointer together and storing it back into
rax\n");
    } // end else.

    // If the identifier is an array, add the contents of rbx and rax into rax.
    if ( p->stmt0 != NULL )
        fprintf( fp, "\tadd rax, rbx\t; adding the internal offset to the regular offset if we encounter an
array\n" );

} // end emitIdentifier function.

// Precondition: The parameter node points to
// an AST node type ARGLIST.
// Postcondition: Each individual argument will
// be evaluated and the answer will be stored in
// the ARGLIST's offset.
void evaluateArguments( FILE *fp, ASTnode *p ) {

    // Stopping case for recursive function. If we reach
    // this case either two cases occured: 1) We traversed
    // through the entire list or there are no arguments
    // to evaluate.
    if( p == NULL )
        return;

    if( mydebug )
       printf( "Evaluating an argument.\n" );

    // Check the argument's type. NOTE: An argument can be an expression, and
    // an expression can be an expression, number, variable, and/or function call.
    switch( p->stmt0->type ) {

       // If the type of stmt0 is NUMBER, store that number into rax.
       case NUMBER:
          fprintf( fp, "\tmov rax, %d\t; moving a number into register rax\n", p->stmt0->value);
          break;
       // If the type of stmt0 is IDENT, we need its memory address. Find the memory address by calling
emitIdentifier
       // and storing the contents of that memory address into RAX.
```

```
      case IDENT:
        emitIdentifier(fp, p->stmt0);
        fprintf( fp, "\tmov rax, [rax]\t; move the contents of the identifier's the memory address into
rax\n");
        break;
      // If the type of stmt0 is EXPR, we need to call emitExpression. After this function ends, the result
will be stored
      // in the expression's offset. Find the contents of this offset and store them into register RAX.
      case EXPR:
        emitExpression( fp, p->stmt0);
        fprintf( fp, "\tmov rax, [rsp + %d]\t; moving to register rax the contents of the expression\n", p-
>stmt0->symbol->offset * DW );
        break;
      // If the type of stmt0 is CALL, call emitFunction. We then call the function. After the function
returns,
      // RAX will already have the value we are looking for.
      case CALL:
        emitFunction( fp, p->stmt0 );
        fprintf( fp, "\tcall %s\t; calling a function\n", p->stmt0->name );
        break;
       default: fprintf(fp, "\t; Unknown target in evaluateArguments.\n");

   } // end switch

   // Store the final evaluated argument into the ARGLIST's offset.
   fprintf( fp, "\tmov [rsp + %d], rax\t; Storing the evaluated argument into ARGLIST's offset\n", p-
>symbol->offset * DW);

   // Since an ARGLIST is a list connected by the "next" pointer, recusively call the function with p-
>next.
   // The base case will stop us from having infinate recursion.
   evaluateArguments( fp, p->next );

} // end evaluateArguments

// Precondition: The first node points to an argument list (ARGLIST)
// while the second node points to the parameters (PARAMLIST) of the relevant function.
// Postcondition: The values of the arguments (which are stored in
// their offsets) are copied into each particular
// parameter's offset.
void copyArgsIntoParameters( FILE *fp, ASTnode * arguments , ASTnode * parameters, int
sizeOfFunction ) {

  if( mydebug )
    printf( "Entering the copyArgsIntoParameters function.\n" );

  // Store the current stack pointer into a trivial register (i.e. rcx in this case).
  fprintf( fp, "\tmov rcx, rsp\t; Store the current stack pointer into a trivial register (i.e. rcx in this
case)\n");
```

```
   // Subtract the size of the function (PLUS 1) in order to determine were the function is located on the
stack
   fprintf( fp, "\tsub rcx, %d\t; Subtract the size of the function (PLUS 1) in order to determine were the
function is located on the stack\n", ( sizeOfFunction + 1 ) * DW );

   // Use a while loop that will copy the values of the arguments into the parameters of the function.
   while( parameters != NULL ) {
      fprintf( fp, "\tmov rax, [rsp + %d]\t; Move the contents of the argument into a temporary
register\n", arguments->symbol->offset * DW );
      fprintf( fp, "\tmov [rcx + %d], rax\t; 'Copy' the contents of rax into the parameter's memory
address\n", parameters->symbol->offset * DW );
      // Traverse through the linked list.
      arguments = arguments->next;
      parameters = parameters->next;
   } // end while

} // copyArgsIntoParameters function.

// Precondition: The node points either to null or a list of
// arguments. These arguments can either be an expression, another
// function call, a number, or a variable.
// Postcondition: The arguments will be evaluated and the arguments
// will be copied into the function's parameters memory locations.
// In adition, RAX will contain the final value that wa evaluated
// when we called teh funtion.
void emitFunction ( FILE *fp, ASTnode *p ) {

   if( mydebug )
      printf( "Entering the emitFunction function.\n" );

   // Check if the call has no arguments.
   if( p == NULL )
      return;

   // Evaluate the arguments. stmt0 points to the the first expression in the argument list.
   evaluateArguments( fp, p->stmt0 );
   // Once all the arguments have been evaluated, copy their final values into the parameters
   // of the appropriate function.
   copyArgsIntoParameters( fp, p->stmt0, p->symbol->fparms, p->symbol->mysize );

   // NOTE: We call the function in the case statements, not here!!!!!

} // end emitFunction function.

// Precondition: The parameter points to either null
// or a global variable.
// Postcondition: Global variable is inizalized as a "common"
// in NASM.
```

```
void printCommon( FILE *fp, ASTnode *p ) {
   if( mydebug )
     printf( "Entering the printCommon function.\n" );

  // If p is null, return.
  if( p == NULL )
    return;

  // Because we can have function declarations, we must
  // check that that the node is type VARDEC.
  if( p->type == VARDEC ) {
    // If the node's value > 0, we found an array.
    if ( p->value > 0)
       fprintf( fp, "\tcommon %s %d\t; common integer array variable\n", p->name, p->value * DW );
    // If the node's value is 0, we found a "regular" integer variable.
    else
       fprintf( fp, "\tcommon %s %d\t; common integer variable\n", p->name, 1 * DW );
  } // end if.

  // Recursively call the function.
   printCommon( fp, p->next);

} // end printCommon function.

// Precondition: The parameter node will be type RETURNSTMT.
// Postcondition: Evaluates the expression that the RETURNSTMT points to.
// If the expression is null, we will not evaulate anything.
// If parameter p is NULL, we will also not evalute anything.
// In both scenerios, we are pirnting the "default" return assembly
// code.
void emitReturn( FILE *fp, ASTnode * p ) {

  if( mydebug )
    printf( "Entering the emitReturn function.\n" );

  // Check if parameter p and stmt0 are null. If any of these are
  // null, we do not have to evaluate any expressions. However, we still
  // need to print the "default" return assembly instructions.
  if( ( p != NULL ) && ( p->stmt0 != NULL ) ) {

    switch( p->stmt0->type ) {
      // If the type of stmt0 is NUMBER, store that number into rax.
      case NUMBER:
        fprintf( fp, "\tmov rax, %d\t; moving a number into register rax\n", p->stmt0->value);
        break;
      case IDENT:
      // If the type of stmt0 is IDENT, we need its memory address. Find the memory address by calling
emitIdentifier
      // and storing the contents of that memory address into RAX.
```

```
          emitIdentifier(fp, p->stmt0);
          fprintf( fp, "\tmov rax, [rax]\t; move the contents of the identifier's the memory address into
rax\n");
          break;
        // If the type of stmt0 is EXPR, we need to call emitExpression. After this function ends, the result
will be stored
        // in the expression's offset. Find the contents of this offset and store them into register RAX.
        case EXPR:
          emitExpression( fp, p->stmt0);
          fprintf( fp, "\tmov rax, [rsp + %d]\t; moving to register rax the contents of the expression\n", p-
>stmt0->symbol->offset * DW );
          break;
        // If the type of stmt0 is CALL, call emitFunction. We then call the function. After the function
returns,
        // RAX will already have the value we are looking for.
        case CALL:
          emitFunction( fp, p->stmt0 );
          fprintf( fp, "\tcall %s\t; calling a function\n", p->stmt0->name );
          break;
        default: fprintf(fp, "\t; Unknown target in evaluateArguments.\n");

      } // end switch

    } // end if.

    // At the end of this, the RAX register will have the value we want to use in another function.
    fprintf( fp,  "\n\tmov rbp, [rsp]\t; FUNC end restore old BP\n" );
    fprintf( fp,  "\tmov rsp, [rsp+8]\t; FUNC end restore old SP\n" );
    // Check if the current function we are visiting is the main funciton.
    if( strcmp( functionName, "main" ) == 0 )
      fprintf( fp,  "\tmov rsp, rbp\t; stack and BP need to be same on exit for main\n" );
    fprintf( fp,  "\tret\t; return to the caller\n" );

} // end printReturn function.

// Precondition: The parameter ASTnode must point to the root of the AST tree.
// Postcondition: All the components of a function will be produced into
// assembly code (e.g. function label, for loops, etc.).
void printBody( FILE *fp, ASTnode *p ) {

    // Declare two trival char "strings" that will allow us to do while/if/else statements.
    char * L1;
    char * L2;

    if( mydebug )
      printf( "Entering the printBody function.\n" );

    // Stopping case. If p equals null, return.
    if( p == NULL )
```

```
    return;

  // Use a swtich statement that will compare the
  // type of nodes we are encountering. NOTE: We
  // are in essence traversing through the ENTIRE
  // abstract syntax tree.
  switch( p->type ) {

    // If we encounter a PARAM type, do not do anything.
    case PARAM:
      break;
    // If we encounter a IDENT type, do not do anything.
    case IDENT:
       break;
    // If we encounter a NUMBER type, do not do anything.
    case NUMBER:
       break;
    // If we encounter a CALL type, first "setup" the call function,
    // then call the function in assembly.
    case CALL:
       emitFunction( fp, p );
       fprintf( fp, "\tcall %s\t; calling a function\n", p->name );
       break;
    // If we encounter a VARDEC type, do not do anything.
    case VARDEC:
       break;
    // If we encounter a RETURNSTMT type, evaluate the expression (if any) and print the return
    // assembly code.
    case RETURNSTMT:
       emitReturn( fp, p );
       break;
    // If we encounter a BLOCK type, recursively call
    // its arguments and compound statement.
    case BLOCK:
      printBody( fp, p->stmt0 );
      printBody( fp, p->stmt1 );
      break;

    // If we encounter a  EXPRSTMT type, evaluate the expression
    // and store its final value into its offset.
    case EXPRSTMT:
       if ( p->stmt0 != NULL ) {
         switch ( p->stmt0->type ) {
            // If the type of stmt0 is NUMBER, store that number into rax.
            case NUMBER:
              fprintf( fp, "\tmov rax, %d\t; moving a number into register rax\n", p->stmt0->value);
              break;
            //If the type of stmt0 is IDENT, we need its memory address. Find the memory address by
calling emitIdentifier
```

```
            // and storing the contents of that memory address into RAX.
            case IDENT:
              emitIdentifier(fp, p->stmt0);
              fprintf( fp, "\tmov rax, [rax]\t; move the contents of the identifier's the memory address into rax\n");
              break;
            // If the type of stmt0 is EXPR, we need to call emitExpression. After this function ends, the result will be stored
            // in the expression's offset. Find the contents of this offset and store them into register RAX.
            case EXPR:
               emitExpression( fp, p->stmt0);
               fprintf( fp, "\tmov rax, [rsp + %d]\t; moving to register rax the contents of the expression\n", p->stmt0->symbol->offset * DW );
               break;
            // If the type of stmt0 is CALL, call emitFunction. We then call the function. After the function returns,
            // RAX will already have the value we are looking for.
            case CALL:
               emitFunction( fp, p->stmt0 );
               fprintf( fp, "\tcall %s\t; calling a function\n", p->stmt0->name );
               break;
            default: fprintf(fp, "\t; Unknown target in exprstmt.\n");
          } // end switch
          // Store the final value into EXPRSTMT's offset.
          fprintf( fp, "\tmov [rsp + %d], rax\t; storing the final value of EXPRSTMT into its offset\n", p->symbol->offset*DW );
      } // end if.
      break;
      // Case ASSIGN. First do the right hand side first, then do the left hand side.
      case ASSIGN:
        // Do right hand side first in order to store the final value into the expression's offset.
        printBody( fp, p->stmt1 );
        // Do left side now. This is because we can have an expression in the left hand side.
        emitIdentifier( fp, p->stmt0 );
        // We know that RAX contains the memory address of the identifier and that
        // the EXPRSTMT's offset contains the final value of the evaluated expression.
        fprintf( fp, "\tmov rbx, [%d + rsp]\t; Storing the final value of the expression into a temporary register\n", p->stmt1->symbol->offset * DW );
        fprintf( fp, "\tmov [rax], rbx\t; Store the value into the identifier's memory address\n" );
        // NOTE: For assignment we must do the right hand side first, then left hand side.
        // For expression, we must do the left hand side first, then the right hand side.
        break;
      // If we encounter FUNDEC type, print the corresponding
      // header that each function must have.
      case FUNDEC :
        // Store the current name of the function into a temporary string variable.
        functionName = p->name;
        // Print that function's label.
        fprintf( fp, "%s:\t; start of function\n", p->name );
```

```
        // If the function is main, we need an extra line of assembly code.
        if( strcmp( p->name, "main" ) == 0 )
          fprintf( fp, "\tmov rbp, rsp \t; SPECIAL RSP to RSB for MAIN only\n" );
        // EVERY function must have the following 5 lines of
        // assembly code.
        fprintf( fp, "\tmov r8, rsp\t; FUNC header RSP has to be at most RBP\n" );
        fprintf( fp, "\tadd r8, %d\t; adjust Stack Pointer for Activation record\n", p->value * DW * -1 );
        fprintf( fp, "\tmov [r8], rbp\t; FUNC header stores old BP\n" );
              fprintf( fp, "\tmov [r8+8], rsp\t; FUNC header stores old SP\n" );
              fprintf( fp, "\tmov rsp, r8\t; FUNC header new SP\n\n" );
        // Print the function's compound statement recursively.
        if( p->stmt1 !=  NULL )
          printBody( fp, p->stmt1 );
        // Print a "dafault" return assembly code for all functions.
        emitReturn( fp, NULL );
        break;
      // If we encounter an IFSTMT (if statement), we need to evaluate the expression, compare the
expression,
      // and jump accordingly depending on the expression.
      case IFSTMT:
         L1 = CreateLabel( ); // End of IF
         L2 = CreateLabel( ); // End of ELSE
          switch( p->stmt0->type ) {
            // If the type of stmt0 is NUMBER, store that number into rax.
            case NUMBER:
              fprintf( fp, "\tmov rax, %d\t; moving a number into register rax\n", p->stmt0->value);
              break;
            // If the type of stmt0 is IDENT, we need its memory address. Find the memory address by
calling emitIdentifier
            // and storing the contents of that memory address into RAX.
            case IDENT:
              emitIdentifier( fp, p->stmt0 );
              fprintf( fp, "\tmov rax, [rax]\t; move the contents of the identifier's memory address into
rax\n" );
              break;
            // If the type of stmt0 is EXPR, we need to call emitExpression. After this function ends, the
result will be stored
            // in the expression's offset. Find the contents of this offset and store them into register RAX.
            case EXPR:
              emitExpression( fp, p->stmt0 );
              fprintf( fp, "\tmov rax, [rsp + %d]\t; moving to register rax the contents of the expression\n",
p->stmt0->symbol->offset * DW );
              break;
            // If the type of stmt0 is CALL, call emitFunction. We then call the function. After the function
returns,
            // RAX will already have the value we are looking for.
            case CALL:
              emitFunction( fp, p->stmt0 );
              fprintf( fp, "\tcall %s\t; calling a function\n", p->stmt0->name );
```

```
            break;
          default: fprintf(fp, "\t; Unknown target in IF/ELSE statement.\n");
        }// end swtich

        // We know that rax contains the value of the evaluated expression. Compare it to 0.
        fprintf( fp, "\tcmp rax, 0\t; comparing register rax to 0. If equal, that means that the expression is false\n" );
        fprintf( fp, "\tje %s\t; IF branch out\n\n", L1 );

        // Recusively call the "body" of the if part. Then jump out of the if's body in order to not do the 'else' part.
        printBody( fp, p->stmt1 );
        fprintf( fp, "\tjmp %s\t; jumping out of the 'if' body\n", L2 );

        // Print label that will allow us to skip the if's body if the expression is false
        fprintf( fp, "\n%s:\t; label that will allow us to skip the if's body if the expression is false\n", L1 );

        // Recusrively call the "else" part.
        if( p->stmt2 != NULL )
          printBody( fp, p->stmt2 );

        // Print the label that will allow us to skip the else part.
        fprintf( fp, "\n%s:\t; Label to exit the else part\n", L2 );
        break;
    // Case READSTMT. This case will allow us to read in a value and store it in a variable's memory location.
    case READSTMT:
      // Make sure that register rax contains the memory address of the
      // specific identifier.
      emitIdentifier(fp, p->stmt0);
      fprintf(fp, "\tGET_DEC 8, [rax]\t; READ in an integer\n");
      break;
    // ITSTMT case (i.e. WHILE statement). If we reach this a ITSTMT type node, we must evaluate the expression
    // and do the while body until the expression becomes false.
    case ITSTMT:
        // Create two labels that will allow us to do a while loop.
        L1 = CreateLabel( );
        L2 = CreateLabel( );

        // Print a label that will be the start of the while loop.
        fprintf( fp, "\n%s:\t; start of while\n", L1 );

        // Use a switch statement that
        switch( p->stmt0->type ) {
          // If the type of stmt0 is NUMBER, store that number into rax.
          case NUMBER:
            fprintf( fp, "\tmov rax, %d\t; moving a number into register rax\n", p->stmt0->value);
            break;
```

```
            // If the type of stmt0 is IDENT, we need its memory address. Find the memory address by calling emitIdentifier
            // and storing the contents of that memory address into RAX.
            case IDENT:
              emitIdentifier( fp, p->stmt0 );
              fprintf( fp, "\tmov rax, [rax]\t; move the contents of the identifier's memory address into rax\n" );
              break;
            // If the type of stmt0 is EXPR, we need to call emitExpression. After this function ends, the result will be stored
            // in the expression's offset. Find the contents of this offset and store them into register RAX.
            case EXPR:
              emitExpression( fp, p->stmt0 );
              fprintf( fp, "\tmov rax, [rsp + %d]\t; moving to register rax the contents of the expression\n", p->stmt0->symbol->offset * DW );
              break;
            // If the type of stmt0 is CALL, call emitFunction. We then call the function. After the function returns,
            // RAX will already have the value we are looking for.
            case CALL:
              emitFunction( fp, p->stmt0 );
              fprintf( fp, "\tcall %s\t; calling a function\n", p->stmt0->name );
              break;
            default: fprintf(fp, "\t; Unknown target in WHILE statement.\n");

          }// end swtich

          // We know that register RAX contains the final evaluated expression. Compare it to 0.
          fprintf( fp, "\tcmp rax, 0\t; comparing register rax to 0. If equal, that means that the expression is false\n" );
          fprintf( fp, "\tje %s\t; while branch out\n\n", L2 );

          // Recusively call the body of the while statement.
          printBody( fp, p->stmt1 );

          // Jump back to check the while's expression.
          fprintf( fp, "\tjmp %s\t; jump back to check expression\n", L1 );
          // If the expression is false, jump to this label to end the while loop.
          fprintf( fp, "\n%s:\t; end of while\n", L2 );
          break;

        // WRITESTMT case. If we ecnounter a WRITE statement we must first check if we are writing a
        // a string. If we are write the string, else evaluate the expression.
        case WRITESTMT:
          // First check if we want to write a string.
          if ( p->stmt0 == NULL ) {
            // Produce assembly code to print the string using its label.
            fprintf( fp, "\tPRINT_STRING %s\t; print a string\n", p->label );
            // Print a newline in the assembly.
```

```
      fprintf( fp, "\tNEWLINE\t; standard Write a NEWLINE\n" );
    } // end if.
    else {
      // This else statement will take care if we want to write an expression.
      switch ( p->stmt0->type ) {
        // We want rsi to have the contents we want to print.
        // If we want to right a number, just move that
        // value into register rsi.
        case NUMBER:
          fprintf(fp, "\tmov rsi, %d\t; moving a number into register rsi\n", p->stmt0->value);
          break;
        // If we encounter a IDENT type, find its memory address
        // and load the contents of that memory address into
        // register rsi.
        case IDENT:
          emitIdentifier(fp, p->stmt0);
          fprintf(fp, "\tmov rsi, [rax]\t; move the contents of the identifier's the memory address into
rsi\n");
          break;
        // If we encounter an EXPR type, evaluate the expression. Then store the final value of
        // the expression (which is found on it's offset) into rsi.
        case EXPR:
          emitExpression(fp, p->stmt0);
          fprintf(fp, "\tmov rsi, [rsp + %d]\t; moving to register rsi the contents of the expression\n",
p->stmt0->symbol->offset * DW );
          break;
        // If we encounter a CALL type, emit function. We know that rax will contain the final value.
Therefore, store the
        // final value into rsi.
        case CALL:
          emitFunction( fp, p->stmt0 );
          fprintf( fp, "\tcall %s\t; calling a function\n", p->stmt0->name );
          fprintf( fp, "\tmov rsi, rax\t; Storing into rsi the contents of rax that was evaluated at the
function call\n" );
          break;
        // Default case. We should NEVER reach this case.
        default: fprintf(fp, "\t; Unknown target write statement.\n");
      } // end switch
      // At the end, rsi will contain the contents we want to print.
      // Use the following two lines of assembly as a general format.
      fprintf( fp, "\tPRINT_DEC 8, rsi\t; standard Write a value\n" );
      fprintf( fp, "\tNEWLINE\t; standard Write a NEWLINE\n" );
    } // end else.
    break;
    // NOTE: We should NEVER hit this case but we put place it in the switch as a "safeguard".
    case ARGLIST:
      break;
    // NOTE: We should NEVER hit this case but we put place it in the switch as a "safeguard".
    case EXPR:
```

```
            break;
      // Default case. We should NEVER reach this case.
      default: fprintf( fp, "\t; NOTE: Unknown type in printBody.\n" );

   } // end switch.

   // Recursively call the function.
   printBody( fp, p->next );

} // end printBody function.

// Precondition: Pointer parameter must point to the abstract
// syntax's tree root node.
// Postcondition: All strings in the abstract syntax tree
// will be produced into assymbly code right after the
// .data section.
void printStrings( FILE *fp, ASTnode *p ) {

   if( mydebug )
      printf( "Entering the printStrings function.\n" );

   // If p is NULL, return.
   if ( p == NULL )
      return;

   // If we encouter a WRITESTMT type,
   // we must print its corresponding label and string
   // into the .data section.
   if( p->type == WRITESTMT ) {
      // Make sure that the write statement writes a
      // string and not an expression.
      if( p->stmt0 == NULL )
         fprintf( fp, "%s: db %s, 0\t; global string\n", p->label, p->name );
    } // end if.

    // Traverse through the ENTIRE abstract syntax tree recusively.
    if( p->next != NULL )
       printStrings( fp, p->next );
    if( p->stmt0 != NULL )
       printStrings( fp, p->stmt0 );
    if( p->stmt1 != NULL )
       printStrings( fp, p->stmt1 );
    if( p->stmt2 != NULL )
       printStrings( fp, p->stmt2 );

} // end printString function.

// Precondition: THE NODE MUST BE AN expresison TYPE.
// NOTE: It cannot be A LEAF SINCE IT WILL GIVE YOU A SEGMENTATION FAULT
```

```
// Postcondition : The result of the expression will be stored in the offset allocated for
// this expression.
void emitExpression ( FILE *fp, ASTnode * p ) {
    if( mydebug )
      printf( "Entering the emitExpression function.\n" );

   // Do the left hand side of the expresison first. Postcondition after this
   // swtich statement: rax must have the expression's final evaluation.
   switch ( p->stmt0->type ){
      case  NUMBER:
        fprintf( fp, "\tmov rax, %d\t; moving a number into the rax register\n", p->stmt0->value );
        break;
      case IDENT:
         emitIdentifier(fp, p->stmt0);
         fprintf( fp, "\tmov rax, [rax]\t; moving the contents of the identifier into rax\n" );
         break;
      case EXPR:
         // If we encounter an expresison, we must recursively call the function.
         emitExpression( fp, p->stmt0 );
         fprintf( fp, "\tmov rax, [rsp + %d]\t; moving the contents of the expression into rax\n", p->stmt0-
>symbol->offset * DW );
         break;
      case CALL:
         emitFunction( fp, p->stmt0 );
         fprintf( fp, "\tcall %s\t; calling a function\n", p->stmt0->name );
         break;
      default: fprintf(fp, "\t; Error: Invalid expression found in left hand side.\n");

   } // end switch.

   // Store the left hand side into the corresponding offset.
   fprintf( fp, "\tmov [rsp +%d], rax\t; storing the left hand side into the offset\n", p->symbol->offset *
DW);

   // We now do the right hand side of the expresison. After this
   // swtich statement rbx must have the right hand side expression's final evaluation.
   switch ( p->stmt1->type ){
      case NUMBER:
        fprintf( fp, "\tmov rbx, %d\t; moving a number into register rbx\n", p->stmt1->value );
        break;
      case IDENT:
        emitIdentifier( fp, p->stmt1 );
        fprintf( fp, "\tmov rbx, [rax]\t; moving the contents of the identifier into rbx\n" );
        break;
      case EXPR:
        emitExpression( fp, p->stmt1 );
        fprintf( fp, "\tmov rbx, [rsp + %d]\t; moving the contents of the expression into rbx\n", p->stmt1-
>symbol->offset * DW );
        break;
```

```
      case CALL:
         emitFunction( fp, p->stmt1 );
         fprintf( fp, "\tcall %s\t; calling a function\n", p->stmt1->name );
         fprintf( fp, "\tmov rbx, rax\t; \n" );
         break;
      // Default case. We should NEVER reach this case.
      default: fprintf( fp, "\t; Error: Invalid expression found in right hand side.\n" );
   } // end switch statement.

   // Fetch the left hand side that was stored in the EXPR's offset temporarlily.
   fprintf(fp, "\tmov rax, [rsp + %d]\t; fetching the left hand side\n", p->symbol->offset * DW);

   // Do the appropriate operation that expression calls for.
   switch( p->operator ) {
      // Add rbx and rax. Store the result back into rax.
      case PLUS:
         fprintf( fp, "\tadd rax, rbx\t; addition expression\n" );
         break;
      // Subtract rbx from rax. Store the result back into rax.
      case MINUS:
         fprintf( fp, "\tsub rax, rbx\t; subtraction expression\n" );
         break;
      // Multiply rax and rbx. Store result back into rax.
      case MULT:
         fprintf( fp, "\timul rax, rbx\t; multiply expression\n" );
         break;
      // Divide rax (the numerator) by rbx (the denominator).
      case DIV:
         fprintf( fp, "\txor rdx, rdx\t; xor rdx to clear the register\n" );
         fprintf( fp, "\tidiv rbx\t; dividing rax by rbx\n" );
         break;
      // Compare rax to see if it is lessthan or equal to rbx (i.e. rax will contain 1 if rax is <= than rbx, 0
otherwise).
      case LESSTHANEQUAL:
         fprintf( fp, "\tcmp rax, rbx\t; EXPR lessthanequal\n");
         fprintf( fp, "\tsetle al\t; sets the last bit in the last byte of the rax register\n");
         fprintf( fp, "\tmov rbx, 1\t; set rbx to one to filter rax\n" );
         fprintf( fp, "\tand rax, rbx\t; filter rax by using the 'and' assembly instruction\n" );
         break;
      // Compare rax to see if it is lessthan rbx (i.e. rax will contain 1 if rax is < than rbx, 0 otherwise).
      case LESSTHAN:
         fprintf( fp, "\tcmp rax, rbx\t; EXPR lessthan\n");
         fprintf( fp, "\tsetl al\t; sets the last bit in the last byte of the rax register\n");
         fprintf( fp, "\tmov rbx, 1\t; set rbx to one to filter rax\n" );
         fprintf( fp, "\tand rax, rbx\t; filter rax by using the 'and' assembly instruction\n" );
         break;
      // Compare rax to see if it is greaterthan rbx (i.e. rax will contain 1 if rax is > than rbx, 0
otherwise).
      case GREATERTHAN:
```

```
            fprintf( fp, "\tcmp rax, rbx\t; EXPR greaterthan\n");
            fprintf( fp, "\tsetg al\t; sets the last bit in the last byte of the rax register\n");
            fprintf( fp, "\tmov rbx, 1\t; set rbx to one to filter rax\n" );
            fprintf( fp, "\tand rax, rbx\t; filter rax by using the 'and' assembly instruction\n" );
            break;
        // Compare rax to see if it is greaterthan or equal to rbx (i.e. rax will contain 1 if rax is >= than rbx,
0 otherwise).
        case GREATERTHANEQUAL:
            fprintf( fp, "\tcmp rax, rbx\t; EXPR greaterthanequal\n");
            fprintf( fp, "\tsetge al\t; sets the last bit in the last byte of the rax register\n");
            fprintf( fp, "\tmov rbx, 1\t; set rbx to one to filter rax\n" );
            fprintf( fp, "\tand rax, rbx\t; filter rax by using the 'and' assembly instruction\n" );
            break;
        // Compare rax and rbx to check if they are equal (i.e. rax will contain 1 if rax and rbx are equal, 0
otherwise).
        case EQUAL:
            fprintf( fp, "\tcmp rax, rbx\t; EXPR equal\n");
            fprintf( fp, "\tsete al\t; sets the last bit in the last byte of the rax register\n");
            fprintf( fp, "\tmov rbx, 1\t; set rbx to one to filter rax\n" );
            fprintf( fp, "\tand rax, rbx\t; filter rax by using the 'and' assembly instruction\n" );
            break;
        // Compare rax and rbx to check if they are not equal (i.e. rax will contain 1 if rax and rbx are not
equal, 0 otherwise).
        case NOTEQUAL:
            fprintf( fp, "\tcmp rax, rbx\t; EXPR notequal\n");
            fprintf( fp, "\tsetne al\t; sets the last bit in the last byte of the rax register\n");
            fprintf( fp, "\tmov rbx, 1\t; set rbx to one to filter rax\n" );
            fprintf( fp, "\tand rax, rbx\t; filter rax by using the 'and' assembly instruction\n" );
            break;
        // Default case. We should NEVER reach this case.
        default: fprintf( fp, "\t; Unknown operator found in operator switch inside emitExpression\n" );

    } // end switch statement.

    // Store FINAL result into the EXPR's offset.
    fprintf(fp, "\tmov [rsp + %d], rax\t; Storing the final answer of the expression into its offset\n", p-
>symbol->offset * DW);

} // end emitExpression function.

// Precondition: Parameter pointer must point to
// the root of the syntax tree. If the parameter pointer is
// NULL, the function will just return.
// Postcondition: The ENTIRE source program is
// converted into assembly code.
void emitAST( FILE *fp, ASTnode *p ) {

    if( mydebug )
        printf( "Entering the emitAST function.\n" );
```

```
  // "Safeguard" statement that will execute if an empty abstract syntax
  // tree is passed as a parameter. This will prevent segmentation faults.
  if( p == NULL )
    return;

  // For every source program, we must print/check for the following components.

  // Printf the include that every sasm assembly code must have.
  fprintf( fp, "\n%%include \"io64.inc\"\n" );
  // Printf the common variables (i.e. the global variables) by calling the
  // printCommon function. Since global variables are connected by "next",
  // we only need to send it the root of the AST.
  printCommon( fp, p );
  // Printf all the .data commonents of the source program. For us, this will
  // include all the strings. NOTE: Commons are not printed in the .data
  // section.
  fprintf( fp, "section .data\t; starting data segment of memory\n" );
  // Call print strings in order to find ALL the strings in the source program.
  printStrings( fp, p );
  // Printf the required .text portion that will start the code segment of memory.
  fprintf( fp, "section .text\t; starting code segment of memory\n" );
  // Every sasm assembly code must have a main function, otherwise an error
  // will occur during compiling time. Make main global.
  fprintf( fp, "\tglobal main\n" );
  // Printf the entire body of the source program (excluding the global variables)
  // by calling the printBody function.
  printBody( fp, p );

} // end ASTemit function.

// Dummy main program so I can compile for syntax error independently.
/*

main( ){

} // end main function.

*/
```