NATIONAL UNIVERSITY OF COMPUTER & EMERGING SCIENCES ISLAMABAD CAMPUS
DATA STRUCTURES - FALL 2021

ASSIGNMENT-2
Due Date: November 07, 2021 (11:59 pm)

Designing an Intelligent Compiler

Overview:
A compiler is a program to process statements written in any programming language and turns into instructions that a computer processor can directly understand. Compiler parses statements using some grammar rules. The CPU interpret instructions by converting the opcode to equivalent microcode.

Example: Working of a C++ program

```
//This is a basic C++ program
int main()
{
   return 0;
}
```

Compiling the above code produces an assembly language code that is close to machine language code. Assembly uses mnemonics to equate the machine code. Once the code is translated to a Machine code it can be directly understood by the CPU and then it is executed with the help of Operating System.

Intelligent Compiler:

Suppose you are designing a compiler for a limited programming language namely ***Recursive Language (RL)*** that is close to C++ and the details of language are provided later in the assignment. **Your task is to compile and simulate the behavior of OS for executing a program consisting of recursion.** Basic working of the Intelligent Compiler is as follows:

1. Reading a text file “code.txt” consisting of a program written in Recursive Language.
2. Compiling the code to understand the actions.
3. Exhibiting the behavior of code by updating the stack segment of a program.
4. Generating necessary output after each step and compiling the final outcome.
5. Reporting the results in “result.txt” file.

Language Model:

The Recursive Language (RL) consists of functions which is a bundle of statements executed together. Components of a function are:

a) Address of each statement
b) Return type in signature
c) Function name in signature
d) Function parameters in signature
e) Function call

RL begins its execution from a "main" function where "main" is a reserved word. Main function appears at the end of code.txt file. Other functions appear before their respective function call in the code.txt file. A function can call another function and even call itself that makes a function recursive in nature.

RL is limited to two primitive data types namely ***int*** and ***float*** for variables. These datatypes can be used as return type or function parameters. Variables can only be created as local variables within the function, and it follows the scope rule of their accessibility and life. Local variables reside inside the Stack Segment of memory. A variable can be declared and initialized in the same line of code. There is no function prototype like in C++. Value is assigned to the variables using the **=** sign and right side of it can have any expression like (A+B*C). The expression can have basic arithmetic operations (+, -, /, *, %) and parenthesis to emphasis the precedence of any operation over others. RL also has comparison operators (<, <=, >, >=, !=, ==) and logical operators (&& and ||). A valid name of variable or function consists of only alphabets, but names are case sensitive means ***Var*** is different than ***var***. The keyword ***print*** is used for writing output to result.txt file and the keyword return is used for returning from the respective copy of function along with data values to the previous function call. The keyword ***void*** can be used as a return type other than ***int*** or ***float***. The quotation sign ***"message"*** is used for string messages and ***\n*** for next line. A function except ***main*** can have any number of parameters and values can only be assigned to the variables at the compile time using the **=** assignment operator.

RL has basic conditional statements (if and if-else) and their compulsory syntax is mentioned below.

```
if (condition)
{
	//body
}
else
{
	//body
}
```

In RL a function may call any other function like **main** function calling function **A**. Therefore, there can be two possible types of recursions:

1. Function A calling another function B and B calling A until the stopping criteria is encountered.
2. Function A calling itself repeatedly.

Sample Program no.1:

```
void Message()
{
        print "This is a recursive function.\n";
        Message();
}
void main()
{
        Message();
}
```

Output no.1:

```
//The above program produces infinite output.
This is a recursive function.
This is a recursive function.
This is a recursive function.
.
.
This is an infinite loop.
```

The Intelligent Compiler is intelligent enough to detect infinite loop and break it by writing "This is an infinite loop" in the result.txt file. It can also report errors in the source code.

Sample Program no.2:

```
void Message(int times)
{
        if (times > 0) // Base case
        {
                print "This is a recursive function.\n";
                Message(times - 1);
```

```
        }
}
void main()
{
        int a=2;
        Message(a);
}
```

Output no.2:

```
//The above program produces infinite output.
This is a recursive function.
This is a recursive function.
```

Sample Program no.3:

```
int sum(int n)
{
        if ( n <= 0 )
        {
                return 0;
        }
        else
        {
                return n + sum(n-1);
        }
}
void main()
{
        print sum(5);
}
```

Output no.3:

```
15
```

Sample Program no.4:

```
int GCD(int x, int y)
{
        if (x % y == 0)
        {
                return y;
        }
        else
        {
                return GCD(y, x % y);
        }
}

int gcd(int x, int y)
{
        if(x > y)
        {
                return GCD(x,y);
        }
        else
        {
                return GCD(y,x);
        }
}
void main()
{
        print gcd(66, 78);
}
```

Output no.4:

```
6
```

Runtime Stack:

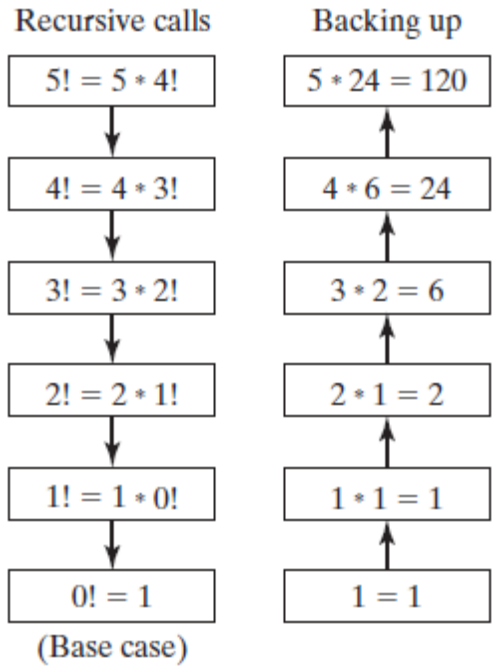
The run time stack is basically the way your programs store and handle your local non-static variables. Think of the run time stack as a stack of plates. With each function call, a new "plate" is placed onto the stack. The local variables and parameters are placed onto this plate. Variables from the calling function are no longer accessible (because they are on the plate below, if stack grows upward). When a function terminates, the local variables and parameters are removed from the stack. Control is then given back to the calling function.

The working of runtime stack for a recursive factorial program is described as follow:

```
C100    int Fact (int n)
C101    {
C102            if (n == 0)
C103            {
C104                    return 1;
C105            }
C106            else
C107            {
C108                    return n * Fact (n-1);
C109            }
C110    }
C111    void main ()
C112    {
C113            print Fact (3);
C114    }
```

- Runtime stack grows in the decreasing order of addresses. ***The structure of node should have a variable to hold address of the node.***
- EBP and ESP registers are used to maintain the runtime stack, where EBP points to the ending of previous frame in stack and ESP points to the top of stack. ***EBP and ESP are two static variables for this assignment. The value of ESP is updated before pushing the next value into the stack.***

- When a Fact (3) is called, 3 will be pushed to the top of the stack which is pointed by ESP. ***Use the push function to insert value N=3 into the runtime stack.***
- The first call to Fact (3) from the main () not just pushed the parameter N=3 but also pushes the return address ***Push(C113)*** from where the function was called. ***Assume code.txt file has associated line numbers as shown above.***
- The diagram shows the recursive function for finding the factorial.



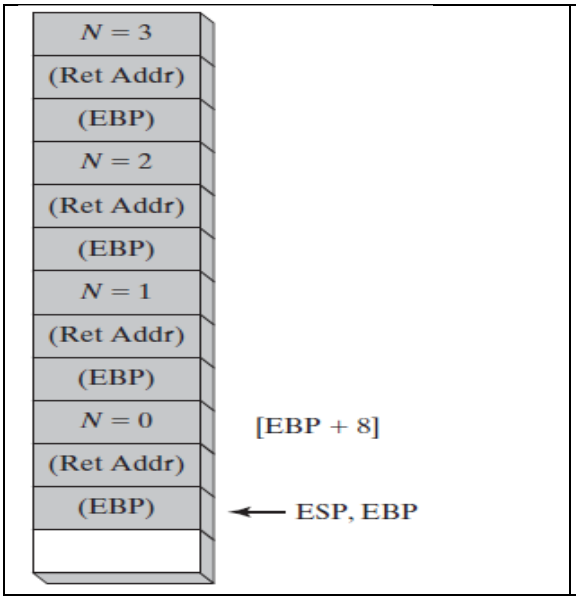
- Next step in the Factorial function call is to Push (EBP) into the stack. ***Remember EBP at this step has address of old ESP value, which is not required in the first function call, so assume it as 1004. After pushing the new value of ESP is copied to EBP variable (not copied into the stack). This copied value will be used later to be pushed.***

```
Factorial PROC
    push  ebp
```

Next, it must set EBP to the beginning of the current stack frame:

```
    mov   ebp,esp
```

- The Fact (3) later calls Fact (2) and same steps of pushing N=2, Return Address and Backing up EBP by pushing are repeated. The below diagram shows the stack frames till Fact (0) is called.



Addresses	*Values in Stack*
1000	N=3
996	C113
992	EBP=1004
988	N=2
984	C108
980	EBP=992
976	N=1
972	C108
968	EBP=980
964	N=0
960	C108
956	EBP=968

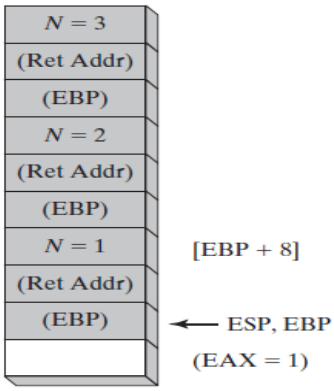
- ***The next step is to copy the value of N in EAX variable. To access the value of N, no need to pop out all other values. Remember in the previous step EBP=ESP. Just add [EBP+8] to access the stored local parameter.*** Note EAX is just a variable, and you may use other variables where required for implementing the logic of recursive function compilation.

> *Tip:* You may have observed that the previous value of EAX, assigned during the first call to Factorial, was just overwritten by a new value. This illustrates an important point: when making recursive calls to a procedure, you should take careful note of which registers are modified. If you need to save any of these register values, push them on the stack before making the recursive call, and then pop them back off the stack after returning from the call. Fortunately, in the Factorial procedure it is not necessary to save the contents of EAX across recursive procedure calls.

- The last call of Fact (0) will start returning. ***Use variable EAX for storing the returned value. EAX must be assigned the factorial's return value. For 0! the returned value is 1 and store it into EAX.***
- ***On returning access the value of N using [EBP+8] and multiply it with EAX. The last return will have 3x2x1x1=6.***

```
ReturnFact:
     mov   ebx,[ebp+8]          ; get n
     mul   ebx                  ; EDX:EAX = EAX * EBX
L2:  pop   ebp                  ; return EAX
     ret   4                    ; clean up stack
```

- ***Next step is to pop a value in the EBP i.e., the updated value of EBP after the last function call will be EBP=968 (remember EBP+8 will be used to access next local variable). Furthermore, pop the return address and pop one more time to remove N=0 from the stack.*** The below diagram shows the updated stack on returning.



Instructions:

1. Your program should read a code.txt file having the source code.
2. Remove extra lines and useless spaces from the source code.
3. Edit the code.txt file to give line number to each line of code. The line number begins from **C100** and the next line will have address **C101**.
4. Identify the main function to begin the execution of the program and also identify other functions in the file.
5. Implement a **runtime stack** using the linked list. The expression involving function calls like ***return n + sum(n-1);*** should only be handled using the runtime stack. Implement the above-mentioned detail of runtime stack. There is only one Runtime Stack to maintain the sequence of function calls along with local variables.
6. The RL may have infix expressions like ***return Fibonacci(n-2)+Fibonacci(n-1)*** or ***(A+B)*C*** and a **Special Stack (SS)** should be used for converting infix expression to postfix. Use an array-based implementation of stack for evaluation of postfix expression. Details of SS are provided below.

   Use a linked list based implementation of Two Queues to exhibit the behavior of Stack (SS) using Queues. Follow these steps to implement (SS).

   1. To push an item into the stack, first move all elements from the first queue to the second queue, then enqueue the new item into the first queue, and finally move all elements back to the first queue. This ensures that the new item lies in the front of the queue and hence would be the first one to be removed.
   2. To pop an item from the stack, return the front item from the first queue.

7. The Node should have the following structure for the runtime stack.

```
class RuntimeNode
{
        int/datatype data;
        RuntimeNode *nodeAddress; //same will be stored in ESP and EBP
                                  //You may modify the highlighted line
        RuntimeNode *next;
}
```

Submission Criteria & Guidelines:

1. Submission: You are required to use Visual Studio 19 or above for the assignment. Combine all your work in one .h file named ROLL_NUM_A_02.h (e.g., 20i-1234_A_02.zip). DO NOT SUBMIT COMPLETE PROJECT. Move .h file to folder as ROLL_NUM_A_02 then .zip file. Submit the .zip file in the classroom within a given deadline. Failure to submit according to the above format would result in ZERO marks.
2. Path of files must be the same as the Project path. DO NOT USE ABSOLUTE PATH.
3. Code must be generic. Try to use template.
4. Use Structure/Class for the linked list implementation.
5. Use Linked List only. Array is not allowed except required for the stack to solve the infix expression.
6. For convenience, using string data type is allowed, but do not use STLs or any built-in functions.
7. If the required output is generated, you will be awarded full marks. Failing to generate the correct output will result in zero marks (black box checking only).
8. For syntax errors, no marks will be awarded.
9. Plagiarism cases will be dealt with strictly. If found plagiarized, both involved parties will be awarded zero marks in this assignment. Copying from the internet is the easiest way to get caught!
10. Deadline: Correct and timely submission of assignment is the responsibility of every student. Therefore, no relaxation will be given to anyone.
11. If a test case is found out to have errors, report to get test cases resolved quickly and updated in the submissions tab with a notice.

Best Wishes!