Kennesaw State

Interpreter for EIL

Ethan Hunt

CS 4308 Concepts of Programing languages

Spring semester 2022

Sharon Perry

3/25/2022

Table of Contents

[**Project Statement:** 2](#_Toc101022025)

[**Interpreter:** 3](#_Toc101022026)

[**Simple Interpretation of Expressions:** 3](#_Toc101022027)

[**Simple Expression Evaluator Code in C++** 4](#_Toc101022028)

[**Complex Interpretation of Statements:** 4](#_Toc101022029)

[**Loop Clock:** 5](#_Toc101022030)

[**Starr Object and Functions** 5](#_Toc101022031)

[**Loop Function Code Example in C++** 8](#_Toc101022032)

[**I/O and Additional Statements:** 8](#_Toc101022033)

[**Input Statement** 8](#_Toc101022034)

[**Output and Code examples:** 9](#_Toc101022035)

[**Function List:** 13](#_Toc101022036)

# **Project Statement:**

The project is 100% complete will all parts working. In addition to stander loops one additional embedded loop has also been implemented.

# **Interpreter:**

My main methodology for the interpreter was to take the parsed tokens and immediately evaluate very statement before moving on tow the next. So in in essence I am merely parsing line by line and then interpreting said line before parsing the following. I again went into designing this blind, so all my methodologies were derived by myself. Since I already had my vision for my Interpreter mapped out from the start and updated my plan as I went, I already had planned for a lot of the simple memory components before the implementation of the parser started. One such example was how since we know that there exists a max of 26 possible variables, I decided to initialize them all at the start of run time and such that they are stored in an array of size 26 called Registers. The names were based on the concept of registers in computer memory such that certain registers can have different purposes set out. The concept of holding specific numbers in the Op code was developed during the scanner phase. Since there exist infinite numbers, there exists a mapping of all lexemes to a unique opcode this includes unique numbers. My original idea was to construct the interpreter using only token values, yet to do that each number must have a specific token value as if all numbers had same token value nothing could be determined without additional information outside of the opcode. To do this I simply shifted the starting point for all numbers up by a set amount. Since -100 is not interpretable in my language as all explicit numbers written inside an expression must be unsinged. However, -100 can be written like 0-100 instead as this follows the axiomatic rules. Thus, the rule of starting zero at some constant above the other token values allows for all numbers below the constant to be tokens of non-numeric lexemes.

## **Simple Interpretation of Expressions:**

Withs EIL’s simple approach to reading expression’s by simply building two arrays one for values of the expression and one for the operators. Then by simply going one by one through each array and preforming a specific case based on the token value of the operator the expression can be evaluated. This is handled by the Exp1 object which handles a given set of op code and can identify its components and evaluate them.

Take of example the following expression a = a \* 2 + 3 – 100 / x ; where a = 1 and x = 1

Save token value of <a> and uses it as the index of the integer stored in <a>. Let this be the value of the index to which we are altering the memory.

Next split the array into values and operators.

The <value\_array> is as follows 1, 2, 3,100, 1 Note: use the tokens values of <a> and <x> as index of the memory array and replaces them with values stored at the specific index so the value array becomes 1,2,3,100,1 instead of a, 2, 3,100, x

The <operator\_array> is as follows \*, + , - , /

After splitting the expression up, we need to evaluate the expression.

First set a int variable named <running\_total> to <value\_array[0]>. This will allow us to have expressions with no operators.

Next create a *for loop* with a count set to the length of the <operator\_array>

Simple iterate through the array and evaluate the <expression>

running\_total = <running\_total> <operator> <value\_array[i]>

in the language of your interpreter. The different operators can be swapped out based on different cases.

After the loop is finished the value of the <running\_total> is inserted into the memory array at the index of the id used at the beginning of the assignment expression.

### **Simple Expression Evaluator Code in C++**

This the code to evaluate an expression:

int Exp1::evalExp1(){

int rt= val[0];

int OPID;

if(l>1){

for(int i=1; i<l;i++){

rt=eval(rt,val[i], OpIDs[i-1] ); }

}

return rt;

}

By simply setting the rt to the initial; value in the expression we do not to need to write separate code to handle single value assignments due to conditional dependence. Then by taking the current rt and the next value in the expression simple calling the *eval*() function will solve the subexpression. The *eval*() looks at the current operate and based on the available cases of possible operators it selects the conditional of the operation and which has a simple expression written in C++ using said operator.

## **Complex Interpretation of Statements:**

Due to the way of evaluating the expressions mention above a critical flaw is created in which is only apparent when looking at how this works under looping conditions. When an expression is made a static copy of the variables value is passed into the expression’s value array upon the parsing process. Only after an iteration of a loop is completed and a second round of parsing is completed in which variables are refreshed and updated in the expression. So, then what’s the problem if the variables are getting updated on the completion of the loop iteration? Well, if I have a loop with multiple statements that use a particular variable as the assign too part of an assignment statement and then another statement calls that same variable again in an expression then the expression will only evaluate using the static copy of the variable it was given at the time of parsing.

### **Loop Clock:**

To circumvent this issue, a method to periodically update the values of the variables in the expression is needed. Thus, the concept of the Loop Clock is born to solve the rate at which the values in the expressions must be refreshed. To refresh the variables the Starr object was created which is a collection of the different statements in the loop with the ability to update its contents easily like an array but using the structure of a stack. Every time each of the loop’s commands are executed the Starr object performs an update to maintain the contents of the expressions. After all commands have been executed in an iteration the queue is reset back to the first Expression in the Starr and this repeats until all iterations have been completed the given set of iterations.

The Loop clock depicted in the graph below shows a loop with 4 statements looped three times

Loop Clock works to both maintain expression variables as well as to tell the order in which statements of any type in the loop gets executed. To implement this simply change the Loop Clock would on certain values of the counter preform other types of operations.

### **Starr Object and Functions**

The object Starr was created to combine the easy of resizing of a stack with the iterative access of an array. It has two main functions one which is a copy of the existing expression evaluator and the other is the *maintain()* function which performs a set of transformations to update the variables in the statements. The *maintain()* has a time complexity of where n is the number of statements.

Example of the *maintain()* process of a 3 statement Loop for one full iteration:

Diagram

Description automatically generated with medium confidence

Code example of the *maintain()* process

void Starr::maintain() {

int i = 0;

int f = 0;

Exp1 current;

int \* Code;

while (i < count) {

Tstack.push(Mstack.top());

Mstack.pop();

i++;

}

current = Mstack.top();

Mstack.pop();

f = current.varID;

current = get\_Expr1(current.Code, current.z, Reg);

current.varID = f;

Reg[f] = current.evalExp1();

Mstack.push(current);

while (!Tstack.empty()) {

Mstack.push(Tstack.top());

Tstack.pop();

}

}

### **Loop Function Code Example in C++**

for (int i = 0; i < loopS; i++) {

if (n % get\_stack\_size(Cstack) == 0) {

n = 0;

}

if (n == get\_stack\_size(Cstack)) {

n = 0;

}

LoopCode.count = n;

LoopCode.maintain();

n++;

}

Registers = LoopCode.Reg;

}

This is what the code looks like for looping an expression this does not show I how I load the statements into the LoopCode variable. Which was simply an instance of an object I created known as Starr. It simple holds and array with a list of specific tokens making up a statement and stores the length of said statement. The loopS variable is used to extend the duration of the loop from the requested count to count \* (# the of statements in loop). Each if statement rests the value n which corresponds to the level in the loop clock cycle. The values of the variables are passed back to the interpreter after each iteration to maintain the integer if a possible runtime failure occurred providing backup Register.

## **I/O and Additional Statements:**

Additional Language and Interpreter functionality has been added as a proof of concept for various other things interpreters can handle. While the simply Output functionalities have been already implemented with auto print and print memory functions in the interrupter, Input functions have been added as well. Random numbers have also been added with limited functionality.

### **Input Statement**

Simple user input was obtained by using the native I/O functionality of C++.

IN <var> <punctuation>

The variable the user selected is then updated with the user input.

**Random Number Statement**

Like the statement structure of Input statement, the Random value statement set as user specified variable equal to a random number by using the random number functionality of C++ to.

RAND <var> <punctuation>

The variable the user selected is then updated with the Random inputs.

## **Output and Code examples:**

\* Note these outputs is correct and verified with by comparing outputs with expected results. If confused on results note that since EIL auto prints out variable changes after an assignment statements execution. This would cause variables used as initializers or expression holders to be printed leading to additional numbers being displayed.

Code for the Fibonacci sequence through i+1 terms

Start M

a = 1 ; b = 0 ; c = 0 ; i = 10 ;

L i

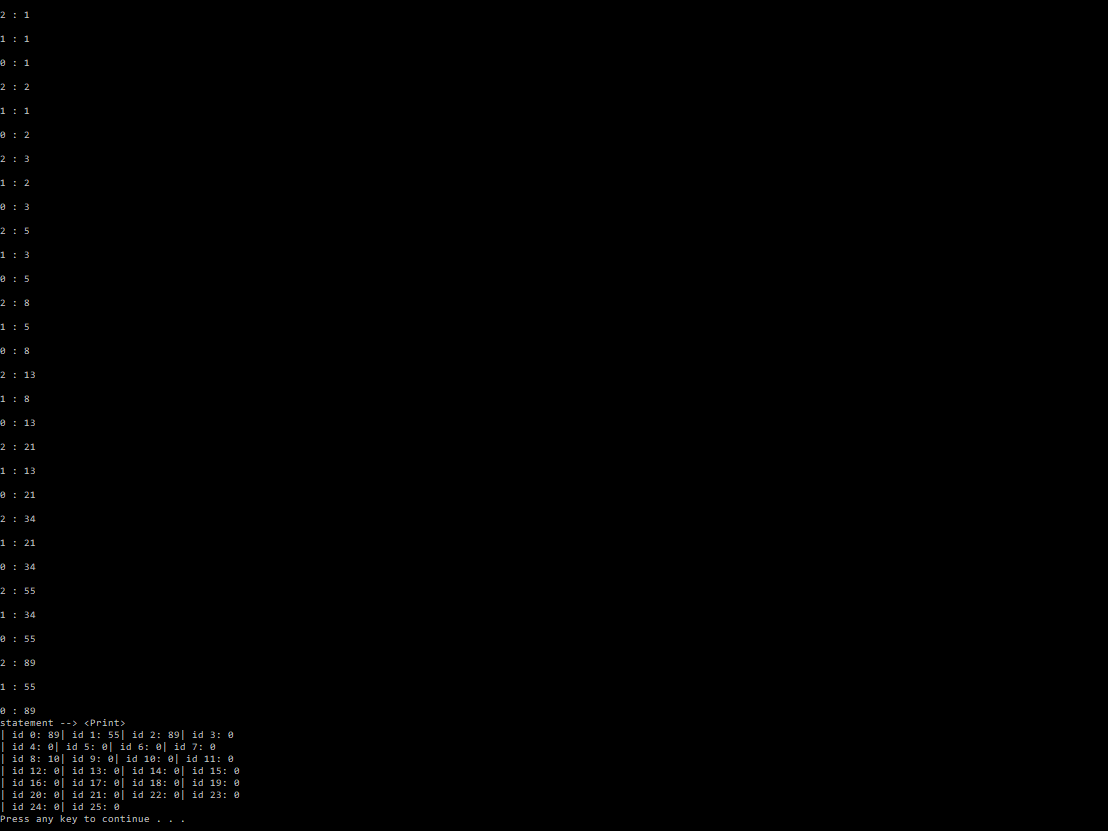
c = a + b : b = a : a = c : ;

P ;

End M

\* Output was too large to display on picture final number in sequence is 89

Graphical user interface, text

Description automatically generated with medium confidence

Code of a Loop Clock written in EIL for a period of 3

Start M

a = 0 ; c = 6 ; b = 3 ;

L c

IF a == b

a = 0 $ :

a = a + 1 : ;

P ;

End M

Text

Description automatically generated

Code for (i+1)!

Start M

i = 5 ; z = 0 ; b = 1 ;

L i

IF a == z

a = 1 $ :

a = a + 1 : b = a \* b : ;

P ;

End M

Text

Description automatically generated

## **Function List:**

Text

Description automatically generated