ZG Script This zg script won't use any third party library.

Contenido

[2. Implementing our script Part 1 4](#_Toc442423120)

[2.1 Defining our first part. 4](#_Toc442423121)

[2.2 AST design and examples 5](#_Toc442423122)

[2.2.1 Definition 5](#_Toc442423123)

[2.2.2 Operator priortity 6](#_Toc442423124)

[2.2.3 Porting assembler code 9](#_Toc442423125)

[2.2.4 Problem with (–) operator and preoperators 11](#_Toc442423126)

[2.2.4 Implementing AST 13](#_Toc442423127)

[2.3 Objects interfacing script 17](#_Toc442423128)

[2.3.1 CObject 17](#_Toc442423129)

[2.3.2 Object operator registration 19](#_Toc442423130)

[2.4 Virtual machine (semi) 24](#_Toc442423131)

[2.4.1 Structure virtual machine (1st version) 25](#_Toc442423132)

[2.4.2 Insert mov instructions 26](#_Toc442423133)

[2.4.3 Insert operator instruction 27](#_Toc442423134)

[2.4.5 virtual machine 1st version 28](#_Toc442423135)

[3. Working with variables 29](#_Toc442423136)

[3.1 Definition 29](#_Toc442423137)

[3.3 Preoperators ++,-- 30](#_Toc442423138)

1. Introduction

This document aims to explain how to build an scripting engine in c++ doing with a sintax like javascript. So for example we can have thinks like that,

**var** i = 0;

This zg script won't use any third party library it means that all things will be done from scratch. It sounds hard but in the end may be it could be funny (I hope). Our script will have thsese features,

syntax like javascript

class with extend support.

The sections that we are going to explain are these,

c++ objects within script

constant scope

AST

# 2. Implementing our script Part 1

This section will describe the basic core for our script engine. We will study this things,

* Defining our first part.
* AST design and examples.
* Porting assembler code
* Implementing our first virtual machine

## 2.1 Defining our first part.

Our first part will integrate only value operations that involves number, string and boolean results with those defined operators as well.

* Number operator: +,\*,-,>>,<<,/,%
* Boolean operator: ==,>,<,>=,<=,&&,||,!=
* String operator: +

So for example we will have operations this,

4+5+6/100 // it gives a number as a result

0>1 // it gives a boolean as a result

"hello there're a "+100+" dogs in my way" // it gives a string as result

## 2.2 AST design and examples

### 2.2.1 Definition

The AST gives a structured way any complex operation subdivided into binary tree until it reaches the most simplified operation. That allows to show a simple way to convert it into assembly code (we will see next section).

For example, these piece of code,

4+5+6

Can be expressed as,

4 +5 +6

E0 E1 E2

E0 + E1 + E2

E3

E3 + E2

E4

E4 (result)

The equivalent AST will be,

### 2.2.2 Operator priortity

The AST gives us a way to subdivide a complex expression in trivial sub expressions but depending of the operator we have to process before or after another sub expressions to get the right result. For example, \* operator has more priority than + or - operator,

5+6\*3

Can be expressed as,

5 +6 \* 3 // We set all number as subexpressions

E0 E1 E2

E0 + E1 \* E2 // Because \* has more priority, we select this before any other

E3

E0 + E2  // Finally, we do the add.

E4

E4 (result)

The AST will be,

If we take account of parenthesis the order of processing it changes. So for example we can have,

(5+6)\*3

First place the operation add will be processed before multiplier operator.

The expression will be processed with levels,

(5 + 6) \* 3 // We process the first level 5+6

E0 E1

(E0 + E1 )\* 3 // Then the parenthesis is solved as E3.

E2  E3

E2 + E3  // Finally, we do the multpliply.

E4

E4 (result)

The AST will be,

So we have to take care about priority in function of operator and the parenthesis have the most priority. Then in numbers, the priority operator is the following,

(),\*, /,%,+, -

### 2.2.3 Porting assembler code

Since we explained how to break down complex expression subdivided into trivial operations with its order of process, we can build the compiler that ports any expression to assembler.

In assembly code we have trivial operations as *multiplication*, *add*, *division*, etc that they are already defined in somewhere. These operations are functions that performs the operations in high level defined by the programmer instead of real assembler codes. So to perform operations in numbers, *String* and *Boolean* we have to define the function operators for +,-,/ and so on. This will be a task that will define in the next section. We suppose that all operators are defined.

So the process is the follow,

1. Process all three nodes until you get the most depth node (the most node is the most priority one). The most depth node can be processed as information to avoid any cost at runtime.

2. Get the operator and process all nodes until its parent == NULL.

So for example for this following AST,

The resulting assembly code will be,

MOV E1,6

MOV E2,3

MUL E3,E1,E2

MOV E0,5

ADD E4,E0,E3

Easy peasy, isn't it ? Each Enrepresent a register. We can define a CPU with a limited registers and use them as soon the register is not used anymore. Because nowadays, we have powerful machines and there's not worry about how many memory the program takes we will define a limited stack called ST in order to resolve expressions that it let us a straightforward way to resolve expressions.

So the same operations we defined in X.XX we can rewrite as,

MOV ST[1],6

MOV ST [2],3

MUL ST [3], ST [1], ST [2]

MOV ST [0],5

ADD ST [4], ST [0], ST [3]

After an expression is processed, the stack can be freed in order to execute another expression. We can define ST within a limit of 128 operable expression. Also we can have a ST for all results.

But how the codes are loaded at runtime. We have to define our objects to process the operations ADD, MUL, ETC. Let's see next section. Each object have to have defined its operators, so for example % operator is not allowed for string.

### 2.2.4 Problem with (–) operator and preoperators

The assembler code work well combined with +,/\* but it fails with series of subs. For example this piece of code fails,

1-2-3

Let’s see the AST,

And the generated assembler will be,

MOV E1,2 // 2

MOV E2,3 // 3

SUB E3,E1,E2 // 2-3 = -1

MOV E0,1 // 1

SUB E4,E0,E3 // 1-(-1) = 0

So we can see that it results 0 but in fact is,

1-2-3 = -4

This happens because the – symbol inverts the sign of second operand.

How we can solve that? Well the answer is simple. Let’s try the same operation with sums.

1+(-2)+(-3)=1-5 = -4

So if we put a series of sums instead of rests we solve the problem. The AST will result as,

Let’s the equivalent code

MOV E1,2 // 2

NEG E3,E1 // -2

MOV E2,3 // 3

NEG E4 //-3

ADD E5,E3,E4 // (-2)+(-3) = -5

MOV E0,1 // 1

ADD E6,E0,E5 // 1+(-5) = -4

#### Preoperators

If any expression has an operator at start we have to treat as an special case like the sub operation. We can these pre operators,

* Boolean: !
* Number: -

The problem of the preoperators are solved in same way as the problem of the subs. We only have to put the operator before affected expression. This is the method:

1. First character is a preoperator (-,!,…) mark as preoperator was found.
2. If

### 2.2.4 Implementing AST

The main idea is start the evaluation and suppose group = 0. And split in left/right. If no operator in group 0 is found try recursive function with group + 1. If no operator is found then the expression will suppose trivial value that can be a value or variable. Find the operator in function of current group and get sub expressions and evaluate left and right sides.

First, we have to define some basic stuff,

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <string>

#define MAX\_EXPRESSION\_LENGTH 8192

using std::string;

typedef struct \_tASTExpression \*PASTExpression;

typedef struct \_tASTOperator \*PASTOperator;

typedef struct \_tASTExpression{

string value;

PASTOperator left,right;

}tASTExpression;

typedef struct \_tASTOperator{

char token;

tASTExpression expr;

}tASTOperator;

enum TYPE\_GROUP{

GROUP\_0=0, // +,-,||

GROUP\_1, // \*/% etc

MAX\_GROUPS

};

bool token\_group2(char c){ return c=='(' || c==')';}

bool token\_group1(char c){ return c=='\*' || c=='/' || c=='%';}

bool token\_group0(char c){ return c=='+' || c=='-';}

bool is\_token(char c){ return token\_group0(c) || token\_group1(c) || token\_group2(c);}

char \* IGNORE\_SPACES(const char \*s){

char \*aux=(char \*)s;

while((\*aux)!=0 && ((\*aux)==' ' || (\*aux)=='\t' || (\*aux)=='\n' || (\*aux)=='\r')) {

aux++;

}

return aux;

}

char \* GET\_END\_WORD(const char \*s){

char \*aux=(char \*)s;

while((\*aux)!=0 && !((\*aux)==' ' || (\*aux)=='\t' || (\*aux)=='\n' || (\*aux)=='\r') && !is\_token(\*aux)) {

aux++;

}

return aux;

}

char \* GET\_CLOSED\_PARENTHESIS(const char \*s){

bool end = false;

char \*aux = (char \*)s;

if(\*aux != '('){

printf("Error first character must be a parenthesis!\n" );

return NULL;

}

int num\_levels=1; // because first is a parenthesis;

do{

aux++;

if(\*aux==')'){

num\_levels--;

}else if(\*aux=='('){

num\_levels++;

}

}while((\*aux) != 0 && num\_levels>0);

return aux;

}

And then implement our AST algorithm,

PASTOperator generateAST(const char \*s, TYPE\_GROUP type\_group=GROUP\_0){

char \*aux=(char \*)s;

char \*start\_value, \* end\_value;

char \*start\_expression,\*end\_expression ;

char value[MAX\_EXPRESSION\_LENGTH]={0}; // I hope this is enough...

tASTOperator \*op=new tASTOperator;

bool theres\_a\_token=false;

aux=IGNORE\_SPACES(aux);

if(\*aux==0){

printf("no expression entry\n");

return NULL;

}

if(type\_group>=MAX\_GROUPS) {

printf("max groups \n");

return NULL;

}

start\_expression=aux;

bool theres\_some\_token=false;

bool end = false;

char \*expr\_op=0;

while(\*aux!=0 && !end){ // searching for operator!

aux=IGNORE\_SPACES(aux);

start\_value=aux;

if(\*aux=='('){ // exp within ()

end\_expression = GET\_CLOSED\_PARENTHESIS(aux);

if(end\_expression==0){return NULL;}

if(\*end\_expression==0){

printf("')' not found\n");

return NULL;

}else{ // advance )

end\_expression++;

}

}else{ // value

end\_expression = GET\_END\_WORD(aux);

}

aux=end\_expression;

aux=IGNORE\_SPACES(aux);

if(\*aux!=0){

expr\_op=aux;

if(is\_token(\*expr\_op)){

theres\_some\_token|=true;

}

switch(type\_group){

case GROUP\_0: end = token\_group0(\*expr\_op);break;

case GROUP\_1: end = token\_group1(\*expr\_op);break;

default: printf(“No token group found”); return NULL;

}

aux++; // advance operator...

}

}

if(!end) {// there's no any operators \"type\_group\"...

if(!theres\_some\_token && \*start\_expression!='('){ // only we have a value (trivial)

op->expr.left=op->expr.right=NULL;

op->expr.value=s; // assign its value ...

return op;

}

else{ // there's a token, so let's perform generating its AST

if(\*start\_expression=='('){

char subexpr[MAX\_EXPRESSION\_LENGTH]={0}; // I hope this is enough...

if((end\_expression-start\_expression-2)> MAX\_EXPRESSION\_LENGTH){

printf("expression too long\n");

return NULL;

}

// copy sub expression

strncpy(subexpr,start\_expression+1,end\_expression-start\_expression-2);

return generateAST(subexpr,GROUP\_0);

}else{

printf("try to generate type\_group+1 expression: %s\n",s);

return generateAST(s, (TYPE\_GROUP)(((int)type\_group)+1));

}

}

}else{ // we found the operator respect of GROUPX so let's put the AST to the left the resulting expression...

char eval\_left[MAX\_EXPRESSION\_LENGTH]={0};

char eval\_right[MAX\_EXPRESSION\_LENGTH]={0};

strncpy(eval\_left,s,expr\_op-s); // copy its left side...

op->expr.left=generateAST(eval\_left,type\_group);

op->expr.right=generateAST(eval\_right,type\_group);

op->token = \*expr\_op;

}

return op;

}

Finally we present funtion that generates assembler code given a AST root node,

int generateAsmCode(PASTOperator op, int & numreg){

int r=0;

if(op==NULL){

return r;

}

if(op->expr.left==NULL && op->expr.right==NULL){ // trivial case value itself...

printf("MOV\tE[%i],%s\n",numreg,op->expr.value.c\_str());

r=numreg;

}else{

int right=0, left=0;

left=generateAsmCode(op->expr.left,numreg);

right=generateAsmCode(op->expr.right,numreg);

r=numreg;

printf("%c\tE[%i],E[%i],E[%i]\n",op->token,numreg,left,right);

}

numreg++;

return r;

}

And the main code is,

int main(int argc, char \* argv[]){

if(argc < 2){

printf("Put expression to evaluate.\n");

printf("\n");

printf("Example:\n");

printf("\n");

printf("5\*6+3");

printf("\n");

printf("\n");

return 0;

}

PASTOperator op=generateAST(argv[1]);

if(op==NULL){ // some error happend!

return -1;

}

else{

int numreg=0;

generateAsmCode(op,numreg);

}

return 0;

}

To compile do this,

g++ main.cpp -o main.exe

To execute demo try this,

main (5+6)\*3

And the output will be,

MOV E[0],5

MOV E[1],6

+ E[2],E[0],E[1]

MOV E[3],3

\* E[4],E[2],E[3]

## 2.3 Objects interfacing script

After we can execute the instruction by its order, we have to detect the type of value and perform its evaluation whether is possible or not. We have to register our objects and some operator in order to process expressions through operators. We have to define a base class to have general behavior for all script object. In the script, at least, we have to embed bool, number and string objects, so we will have:

* CObject: Is the base of all operators.
* CNumber: Is going to have the operators +,-,%,/,\*
* CBoolean: Is going to contemplate the operators ||, &&, >,<,>=,<=
* CString: Is going to have the operator +

### 2.3.1 CObject

CObject is the base of all objects. Through CObject we can define base attributes like its name, base functions as string conversion. So let's to define the basic skeleton structure,

public CObject{

public:

CObject();

CObject(const string & str); // construct value by its string…

virtual string toString();

}

#### CNumber

CNumber will contemplate the operations within numbers expressions. The operations that will support are +,/,-,% and \*. Let's define the skeleton structure of CNumber,

class CNumber: public CObject {

public:

static CNumber \*Parse(const string & s);

double value;

CNumber();

CNumber (const string & str\_value);

virtual string toString();

};

#### CBoolean

CBoolean will contemplate the operations within boolean expressions. The operations that will support are <,>,<=,>=,&& and ||. Let's define the skeleton structure of CBoolean,

class CBoolean: public CObject {

public:

static CBoolean \*Parse(const string & s);

bool value;

CBoolean ();

CBoolean (const string & str\_value);

virtual string toString();

};

#### CString

CString will contemplate the operations within string expressions. The operations that will support is only concatenate function +. Let's define the skeleton structure of CString,

class CString: public CObject{

public:

string value;

CString ();

CString (const string & str\_value);

virtual string toString();

};

### 2.3.2 Object operator registration

The last section we explained the objects we will have. We still need to have a way to link the operations with function objects as operators, so we have to have a way to register our object and function interfaces.

#### Register Object

To register a object we have to have a register function that register the name’s object. Let's define a function that registers the objects in first place. We are gona to have factories, Like the name defines it, a factory is the way to have a set of controlled objects so we have the control of all objects in memory. For example, we can get the object using ( for example get(id\_name\_object)). Furthermore, we can set some usefull attribs in order to define object at 100%.

**template**<class \_T>

**void** registerFunction(const string & \_name){

**if**(m\_mapContainerObjects.count(\_name) == 0){ // don’t exist

tObjectInfo info;

info.signature= **typeid**(**\_T \***).name()

\_T::setObjectInfo(**typeid**(**\_T \***).name())

m\_mapContainerObjects)[\_name]=info;

}

}

typedef struct {

string pointer\_type;

string result\_type;

vector<string > \*param\_type;

}tObjectInfo;

And then in some part of our initialization we have to call these functions that register our basic types,

// registerObject registere object to get ready to use into our script.

registerObject<CNumber>("CNumber");

registerObject<CBoolean>("CBoolean");

registerObject<CString>("String");

#### Register functions

Let's register operator for each object functions through lambda functions. We achieve the register through this helper function,

**template** <std::size\_t...> **struct** index\_sequence {};

**template** <std::size\_t **N**, std::size\_t... **Is**> **struct** make\_index\_sequence : make\_index\_sequence<**N**-1, **N**-1, **Is**...> {};

**template** <std::size\_t... **Is**> **struct** make\_index\_sequence<0, **Is**...> : index\_sequence<**Is**...> {};

**template** <size\_t **argIdx**, **typename** **R**, **typename**... **Args**>

**auto** **getArgTypes**(std::string& ref, std::vector<std::string> & params)

-> **typename** std::enable\_if<**argIdx** == 1>::type

{

**using** fun = function\_traits<std::function<**R**(**Args**...)> >;

params.push\_back(std::to\_string(**argIdx**));

// number parameter ...

}

**template** <size\_t **argIdx**, **typename** **R**, **typename**... **Args**>

**auto** **getArgTypes**(std::vector<std::string> & params)

-> **typename** std::enable\_if<**argIdx** != 1>::type

{

**using** fun = function\_traits<std::function<**R**(**Args**...)> >;

params.push\_back(std::to\_string(**argIdx**));

getArgTypes<**argIdx** - 1, params,**R**, **Args**...>(ref);

}

**template** <**typename** **F**, std::size\_t... **Is**>

std::vector<std::string> \* **getParamsFunction**(**int** i, index\_sequence<**Is**...>)

{

std::vector<std::string> \*typeParams= **new** std::vector<std::string>();

getArgTypes<**F**::arity, **typename** **F**::result\_type, **typename** **F**::**template** arg<**Is**>::type...>(s,\*typeParams);

**return** typeParams;

}

**template**<**class** **FunctorT**>

**void**\* **getcodeptr**(**const** **FunctorT**& f) {

**auto** ptr = &**FunctorT**::operator();

**return** \*(**void**\*\*)&ptr;

}

bool registerOperatorInternal(const string & \_op\_name, const string & result\_type,vector<string> \* param\_type, **void**(\*fun\_ptr)()){

if(param\_type->size()!=2){

(“fatal error. We expected 2 ops for operator”); return false;

}

**if**(m\_mapContainerObjects[\_op\_name].operator(\_op\_name) == 0){ // exist

if(!existOperatorSignature(\_op\_name,result\_type, param\_type)){

printf\_error(“operator signature already exist”);

return false;

}

tInfoObjectOperator info;

info.result= result\_type;

info.params = param\_type;

info.fun\_ptr = fun\_ptr;

// IMPPORTANT! for each operator we can have different signatures…

m\_mapContainerObjects[\_op\_name].push\_back(info);

return true;

}

else{

printf(\”operator %s doesn’t exist\”,\_op\_name.c\_str())

}

}

tInfoObjectOperator \* getOperatorInfo(string op, CObject \*op1, CObject \*op2){

string ps1=op1->getPointerNameType();

string ps2=op2->getPointerNameType();

vector< tInfoObjectOperator > v=& m\_mapContainerObjects.count(op);

for(int i = 0;i < v.size(); i++){ // for all signatures operator…

if(v->at(i)->params->size()==2){

if(v->at(i)->params->at(0)==ps1 && v->at(i)->params->at(1)==ps2){ // we found the signature

return v->at(i);

}

}else{

Printf(“fatal error. We expected 2 ops for operator”);

}

}

}

**#define** registerOperator(op\_name,f){\

**auto** reg\_function=f;\

**typedef** function\_traits<**decltype**(reg\_function)> decl;\

std::vector<string> \*param\_type=getParamsFunction<decl>(0,make\_index\_sequence<decl::arity>{});\

**std::string result\_type=typeid**(decl::result\_type).name() << endl;\

registerOperatorInternal(obj\_name,op\_name,result\_type,param\_type, (**void**(\*)()) getcodeptr(asm\_caller));

}

Let’s register the operations,

##### CNumber

// Define operations for CNumber.

// registers operator + to CNumber object.

registerOperator("+",[](CNumber \*n1,CNumber \*n2){

CNumber \*result=new CNumber();

result->value = n1->value + n2->value;

return result;

});

// registers operator - to CNumber object.

registerOperator("-",[](CNumber \*n1,CNumber \*n2){

CNumber \*result=new CNumber();

result->value = n1->value - n2->value;

return result;

});

// registers operator / to CNumber object.

registerOperator("/",[](CNumber \*n1,CNumber \*n2){

CNumber \*result=new CNumber();

result->value = n1->value / n2->value;

return result;

});

// registers operator \* to CNumber object.

registerOperator("\*",[](CNumber \*n1,CNumber \*n2){

CNumber \*result=new CNumber();

result->value = n1->value \* n2->value;

return result;

});

// registers operator % to CNumber object.

registerOperator("%",[](CNumber \*n1,CNumber \*n2){

CNumber \*result=new CNumber();

result->value = n1->value % n2->value;

return result;

});

#### CBolean

// Define operations for CBoolean.

// registers operator < to CBoolean object.

registerOperator("<",[]( CBoolean \*n1, CBoolean \*n2){

CBoolean \*result=new CBoolean ();

result->value = n1->value < n2->value;

return result;

});

// registers operator > to CBoolean object.

registerOperator(">",[]( CBoolean \*n1, CBoolean \*n2){

CBoolean \*result=new CBoolean ();

result->value = n1->value < n2->value;

return result;

});

// registers operator <= to CBoolean object.

registerOperator("<=",[]( CBoolean \*n1, CBoolean \*n2){

CBoolean \*result=new CBoolean ();

result->value = n1->value <= n2->value;

return result;

});

// registers operator >= to CBoolean object.

registerOperator(">=",[]( CBoolean \*n1, CBoolean \*n2){

CBoolean \*result=new CBoolean ();

result->value = n1->value >= n2->value;

return result;

});

// registers operator && to CBoolean object.

registerOperator("&&",[]( CBoolean \*n1, CBoolean \*n2){

CBoolean \*result=new CBoolean ();

result->value = n1->value && n2->value;

return result;

});

// registers operator || to CBoolean object.

registerOperator("||",[]( CBoolean \*n1, CBoolean \*n2){

CBoolean \*result=new CBoolean ();

result->value = n1->value || n2->value;

return result;

});

// Define operations for CBoolean passing CNumber objects.

// registers operator < to CBoolean object.

registerOperator("<",[]( CNumber \*n1, CNumber \*n2){

CBoolean \*result=new CBoolean ();

result->value = n1->value < n2->value;

return result;

});

// registers operator > to CBoolean object.

registerOperator(">",[]( CNumber \*n1, CNumber \*n2){

CBoolean \*result=new CBoolean ();

result->value = n1->value < n2->value;

return result;

});

// registers operator <= to CBoolean object.

registerOperator("<=",[]( CNumber \*n1, CNumber \*n2){

CBoolean \*result=new CBoolean ();

result->value = n1->value <= n2->value;

return result;

});

// registers operator >= to CBoolean object.

registerOperator(">=",[]( CNumber \*n1, CNumber \*n2){

CBoolean \*result=new CBoolean ();

result->value = n1->value >= n2->value;

return result;

});

#### CString

// Define operations for CString.

// registers operator + to CString object.

registerOperator("+",[]( CString \*n1, CString \*n2){

CString \*result=new CString ();

result->value = n1->value + n2->value;

return result;

});

// registers operator + to CString object.

registerOperator("+",[]( CString \*n1, CNumber \*n2){

CString \*result=new CString ();

result->value = n1->value + n2->toString();

return result;

});

So we have all stuff initialized to get the assembler linked within object interface. So we will make a parser to set every variable or constant value and push it to the expression stack.

CObject \*obj;

if((obj=CNumber::Parse(value))!=NULL){ // integer

}else if((obj=CBoolean::Parse(value))!=NULL){ // boolean

} else if(value[0]==\” && value[value.length-1]==”\””){ // string

String s = value.substring(value[1],value[value.length-2]);

Obj=new CString(s);

}else{ // is a conventional variable ?

}

MOV [x],Y

Will be

push\_const\_number(MOV,Y);

push\_const\_string(MOV,Y);

push\_const\_bool(MOV,Y);

Soo we have everything to be registered,

// after you register all operators, try design MOV, other operator as,

Switch(op){

Case MOV: E[i]=CONST\_DATA[i]; break; // can be number, Booleans or strings…

Case OP: E[i]=OP[h](E[j],E[k]); // search for any signature that matches E[j](resident var),E[k](foregner var that tries join to the party)

Case FUN: think about it!ª

}

## 2.4 Virtual machine (semi)

This section is going to explain our first version of the virtual machine. The virtual machine is going to execute expressions in its processed order. The main idea process AST and convert it as linear expression arrays for each sentence or instruction. Before that we have to define these things:

* Structure instruction machine code (1st version)
* Insert **mov** instructions
* Insert operator instructions
* Operator priority
* AST to linear operation

### 2.4.1 Structure virtual machine (1st version)

Our first version of virtual machine structure will have initialized array with all info needed to execute operations. At the moment will have one single statement to make more clear the code operation but next section we will design a code with list of statements

**enum**{

*MOV\_VALUE*=1,

*OPERATOR*

};

**typedef** **struct**{

string result\_type;

vector <string> \* param\_type;

**void** (\* fun\_ptr)();

}tInfoObjectOperator;

**class** tInfoAsmOp{

**public**:

**int** type\_op;

tInfoObjectOperator \*funOp;

CObject \*res;

string type\_res;

**int** index\_left,index\_right;

**bool** (\* isconvertable)(**int** value);

**tInfoAsmOp**(){

isconvertable=NULL;

type\_op=0;

funOp=NULL;

res=NULL;

type\_res="none";

index\_left=index\_right=-1;

}

};

**typedef** **struct**{

vector<tInfoAsmOp \*> asm\_op;

}tInfoStatementOp;

tInfoStatementOp /\*vector<tInfoStatementOp>\*/ statement\_op;

### 2.4.2 Insert mov instructions

The insert mov instruction basically have to detect the type of value and create the object.

**void** **CZG\_Script::insertMovInstruction**(**const** string & v){

CObject \*obj;

// try parse value...

**if**((obj=CNumber::*Parse*(v))!=NULL){

print\_info\_cr("%s detected as number\n",v.c\_str());

}

**else** **if**(v[0]=='\"' && v[v.size()-1]=='\"'){

string s=v.substr(1,v.size()-2);

CString \*so=NEW\_STRING();

so->m\_value=s;

obj = so;

print\_info\_cr("%s detected as string\n",v.c\_str());

}

**else** **if**((obj=CBoolean::*Parse*(v))!=NULL){

print\_info\_cr("%s detected as boolean\n",v.c\_str());

}**else**{

print\_error\_cr("ERROR: %s is unkown variable\n",v.c\_str());

}

**if**(obj != NULL){

tInfoAsmOp \*asm\_op = **new** tInfoAsmOp();

asm\_op->res=obj;

asm\_op->type\_res=obj->getPointerClassStr();

**if**(asm\_op->res->getPointerClassStr()==""){

print\_error\_cr("unknown type res operand");

**return**;

}

asm\_op->type\_op=*MOV\_VALUE*;

statement\_op.asm\_op.push\_back(asm\_op);

}

}

### 2.4.3 Insert operator instruction

The insert operator instruction basically has to find the operator and chec

**void** **CZG\_Script::insertOperatorInstruction**(**const** string & op, **int** left, **int** right){

**if**(left <0 || (**unsigned**)left >= statement\_op.asm\_op.size()){

print\_error\_cr("ERROR: left operant is out of internal stack (%i,%i) ",left,statement\_op.asm\_op.size());

**return**;

}

**if**(right <0 || (**unsigned**)right >= statement\_op.asm\_op.size()){

print\_error\_cr("ERROR: right operant is out of internal stack (%i,%i)",right,statement\_op.asm\_op.size());

**return**;

}

tInfoAsmOp \*asm\_op = **new** tInfoAsmOp();

asm\_op->type\_op=*OPERATOR*;

asm\_op->index\_left = left;

asm\_op->index\_right = right;

**if**((asm\_op->funOp=getOperatorInfo(op,

statement\_op.asm\_op[left]->type\_res,

statement\_op.asm\_op[right]->type\_res)) != NULL){

asm\_op->type\_res = asm\_op->funOp->result\_type;

statement\_op.asm\_op.push\_back(asm\_op);

}**else**{

print\_error\_cr("cannot find operator \"%s\"",op.c\_str());

}

}

### 2.4.5 virtual machine 1st version

The Our first version of virtual machine will execute basically the operations we inserted.

**Void execute**(){

**if**(statement\_op.asm\_op.size()>0){

**for**(**unsigned** i = 0; i <statement\_op.asm\_op.size(); i++){

**switch**(statement\_op.asm\_op[i]->type\_op){

**case** *MOV\_VALUE*: // do nothing because the value is already stored in the list.

print\_info\_cr("mov!");

**break**;

**case** *OPERATOR*:{ // we will perform the operation (cross the fingers)

print\_info\_cr("operator %p",statement\_op.asm\_op[i]->funOp->fun\_ptr);

CObject \*i1,\*i2;

i1=statement\_op.asm\_op[statement\_op.asm\_op[i]->index\_left]->res;

i2=statement\_op.asm\_op[statement\_op.asm\_op[i]->index\_right]->res;

**int** fun=(**int**)statement\_op.asm\_op[i]->funOp->fun\_ptr;

**int** result=0;//(int)statement\_op.asm\_op[statement\_op.asm\_op[i]->index\_left]->res;

**#ifdef** \_WIN32

**asm**(

"push %[p2]\n\t"

"push %[p1]\n\t"

//"push %%esp\n\t"

"call %P0\n\t" // call function

//"add $4,%%esp" // Clean up the stack.

: "=a" (result) // The result code from puts.

: "r"(fun),[p1] "r"(i1), [p2] "r"(i2));

**#endif**

**if**(result!=0){

**if**(**dynamic\_cast**<CObject \*>((CObject \*)result) == NULL){

print\_error\_cr("Error casting object");

**return**;

}

statement\_op.asm\_op[i]->res=(CObject \*)result;

}**else**{

print\_error\_cr("Error result returning void");

**return**;

}

}

**break**;

}

}

CObject \*obj = statement\_op.asm\_op[statement\_op.asm\_op.size()-1]->res;

CNumber \*num;

CString \*str;

CBoolean \*bol;

**if**((num = **dynamic\_cast**<CNumber \*>(obj))!=NULL){

print\_info\_cr("Number with value=%f",num->m\_value);

}**else** **if**((str = **dynamic\_cast**<CString \*>(obj))!=NULL){

print\_info\_cr("String with value=\"%s\"",str->m\_value.c\_str());

}**else** **if**((bol = **dynamic\_cast**<CBoolean \*>(obj))!=NULL){

print\_info\_cr("Boolean with value=%i",bol->m\_value);

}

}

}

Vector<t>

If we remember the code that generates the asm code (figure X.X)

int generateAsmCode(PASTOperator op, int & numreg){

int r=0;

if(op==NULL){

return r;

}

if(op->expr.left==NULL && op->expr.right==NULL){ // trivial case value itself...

printf("MOV\tE[%i],%s\n",numreg,op->expr.value.c\_str());

r=numreg;

}else{

int right=0, left=0;

left=generateAsmCode(op->expr.left,numreg);

right=generateAsmCode(op->expr.right,numreg);

r=numreg;

printf("%c\tE[%i],E[%i],E[%i]\n",op->token,numreg,left,right);

}

numreg++;

return r;

}

# 3. Working with variables

This section will introduce the concept of variable. We will declare variables with instruction var . We are going to explain these points:

1. Definition
2. Operators ++ and –

## 3.1 Definition

So for example if we want to initialize a variable called ***i*** we have to proceed in this way,

**var** i;

Any declared variable is initialized with undefined state that specifies that a variable has been declared without been initialized. So for example,

**var** i=0;

In the example, the variable i has been initialized as number object with value 0. Si we have to define the undefined object.

Class CUndefinedObject: public CObject{

}

In any time, the compiler detects undefined object in any operation, it cancels execution and tells the error.

## 3.2 Declare variable and defining operator =

In order to parse a valid variable , we have to test whether symbol **var** exist in expression. Furthermore, if there’s an operator = then the variable will be initialized as a result of expression. We know how evaluate expressions, so the result of evaluation will be tell the type of instanced variable.

For example, let’s see an example of instanced. The following statement,

**var** i = 3;

Will result the following AST,

Expression

And the equivalent assembler will be,

MOV V0,3

Another example,

**var** i = 3+4;

Will become,

Expression

And the equivalent assembler,

MOV E0,3

MOV E1,4

ADD V0,E0,E1

And that’s it! So the operator = is simple tan assign the resulting expression to variable.

## 3.3 Pre operators ++,--

The pre operators ++ and -- only affects to number object variable and they are executed always before other operations. As an example we can find a simple operation like this,

++i;

For this expression,

10+++j;

The equivalent AST will be,

Expression

And the assembler code will be,

MOV E0,10

INC V0

ADD E1,E0,V0

Another example,

2+3+(6+++k)\*++k;

The equivalent AST will be,

Expression

And the assembler code will be,

MOV E0,6

INC V0

ADD E1,E0,V0

INC V0

MUL E2,E1,V0

MOV E3,3

ADD E4,E2,E3

MOV E5,2

ADD E6,E5,E4

## 3.3 Post operators ++,--

The post operators ++ and -- is performed always after the expression is executed. As an example we can find a simple operation like this,

i++

For this expression,

10+j++;

The equivalent AST will be,

Expression

And the assembler code will be,

MOV E0,10

ADD E1,E0,V0

INC V0

Another example,

2+3+(6+k++)\*k++;

The equivalent AST will be,

Expression

And the assembler code will be,

MOV E0,6

ADD E1,E0,V0

INC V0

MUL E2,E1,V0

MOV E3,3

ADD E4,E2,E3

MOV E5,2

ADD E6,E5,E4

INC V0

## 3.4 Modifying AST for variable support

To have built in variable support with the explications examples we have seen on the last sections, we have to have variable register detection to take account into our AST. Furthermore, we have to take account the pre-operators and post-operators they don't mix with + and – operators together.

3.4.1 Register a variable

As we already know, to register variable we have to detect **var** word in any statement. We can implement the following function,

Bool isVarDeclarationStatment(string & var\_name, PASTNode \*expression);

The function will process only two arguments var\_name that will save the variable name and the str\_expression is related with the AST node that our AST algorithm was detected. The function will return the char with the next valid statement (after ;). If something was wrong, then a NULL character will be returned.

3.4.2 multi statement support

As far as we know, our AST algorithm only process one statement. We have to redefine the statement as a vector of statement in order to process each one.

3.4.3 Pre and post operators

Pre operators

To detect the pre operators we need to know whether there’s a clear two + symbols from its right and its left. However, we can find these situations,

10+++i;

That’s it can treated as 10+(++i). This is clear because 10 is constant. The following expression is hard to determine,

i+++i

Because this can be split with two,

i+(++i) or (i++)+i

What could we do in this situation?

In any case our AST will choose (i++)+i because the algorithm will detect the operator ++ that it has more priority than + operator.

We have to modify our already implemented preoperator\_token function like this,

// detection ++ operator.

**if**(\*aux=='+'){

aux++;

aux=IGNORE\_SPACES(aux);

**if**(\*aux=='+'){

aux++;

aux=IGNORE\_SPACES(aux);

}

**if**(\*aux=='+'){ // is not a valid preoperator

**return** 0;

}

**return** aux;

}

// detection -- operator.

**if**(\*aux=='-'){

aux++;

aux=IGNORE\_SPACES(aux);

**if**(\*aux=='-'){

aux++;

aux=IGNORE\_SPACES(aux);

}

**if**(\*aux=='-'){ // is not a valid preoperator

**return** 0;

}

**return** aux;

}

Post operators

To detect post operators the algorithm has to detect if there’s – or ++ after a non operator. The idea is take advantage of using preoperator function with postoperator function as well. So when the first preoperator bool is put false, then

Supporting variables in virtual machine

To support variables in our virtual machines, we will define a vector called V[X] that means the variable number declared. So in our expressions we can manage E[X] as well as V[X] without problem.

PostEvaluation Operators

The algorithm only has to check that after ++ or – there’s no value like 5,6,7.

We have to define a undefined because this

The idea is that when the script detect a value as number, it will create a CNumber object automatically when in some number is loaded to perform the expression.

The operators are registered in vector internally in some map that can be take out doing that,

operator["CNumber"]["\*"]=function\_info

The function info it tells the function of lambda function and its number of parameters and the type of arguments.

Constant values detection

C++ objects within script

The first step before implement features of our script is implement a way to interface script variables to our c++ code. Let define CObject as a base of all supported objects,

class CObject{

};

Our script will support boolean, string and number operations, so we are going to define a CNumber, CBoolean and CString objects that extends from CObject,

class CNumber: public CObject{

public:

CNumber(const string & s); // construct cnumber parsing input var.

};

class CBoolean: public CObject{

public:

CBoolean(const string & s); // construct cnumber parsing input var.

};

class CString:public CObject{

public:

CString(const string & s); // construct cnumber parsing input var.

};

We suppose class functions are trivial to implement and we are not going to explain its implementation.

The min work to do is presented in the following points:

Define language

Building parser

Assembler

Byte code

Define Language

I chosed a language like JAVA or JAVASCRIPT because for me is more close and readable than other script languages. This section it will define the sintax of our language.

Let’s show some of script examples,

Comments

We will have two types,

Single comment

*// this is a comment*

Block comment

*/\**

*\* This is a block*

*\*/*

Vars

Declare vars,

var i; // declare variable (it remains in undefined state until some value is assigned)

var j = 0; // declare an initialized var.

Function

**function** add(i,j){

**return** i+j;

}

Loops

While

**while**(i<10){

// do something ...

}

**do**{

// do something ...

}**while**(i<10);

For

**for**(**var** i = 0**;** i **<** 10; i++){

// do something ...

}

Scopes

The scope are two:

Global scope

Local scope

Global scope

Is the place where global variables are defined outside any function.

var global\_var=1;

function fun1(arg1, arg2){

global\_var =2; // accessing global var…

}

Local scope

The local scope is the place where the variables are defined.

function Fun1(arg1,arg2){

var Local\_var =1;

}

Scope depth

You can define as many scopes you want with { }. Inside you can declare the same variable name as declared in upper scopes.

function Fun1(arg1,arg2){

var Local\_var =1; // first instance (local\_var is 1)…

{

var local\_var=2; //new instance (local\_var is 2)…

{

var local\_var=3; // new instance (local\_var is 3)…

}

Print(local\_var); // local\_var is 2

}

Print (local\_var); // local\_var is 1

}

So, when the programmer defines {} a new stack is defined to take care local first, and then search for upper scopes.

Expressions

We can split two types expressions,

* Operable expression
* Loop, conditional expression

Statment

Any statement expressions starts after scope ( { ) and ends with semicolon (;). In every expression we can combine operators or returning functions but at the end the expression must deduce the type of resulting object. The evaluation direction for expressions in our script engine will be **left to right**. So, let’s see some expression and its deduced object.

**var** b=10 > 0; // will be a Boolean expression, so the variable will be instanced as Boolean object.

**var** i=10\*10.0\*(5+6); // the expression will be a number, so the variable will be instancied as number.

**var** s=”The dog had ”+100+” owners before his dying.”; // the variable will be instanced as string.

We can use regular expression to build our AST (Abstract Syntax Three)

Two expressions evaluation (E *operator* E)

* Number operator =>[+,\*,-,>>,<<,/,%]
* Boolean operator=>[==,>,<,>=,<=,&&,||,!=]
* String operator=>[+]
* CObject operator=>[.]

One expression **prefix** evaluation (*operator* E)

* Number operator=>[++,--,-,+]
* Boolean operator=>[!,~]

One expression **postfix** evaluation (E *operator*)

* 1 Number operator=>[++,--]

This is an special operator to take account after the expression is evaluated. The resulting expression of postincrement ++ is,

I=J++;

J=J+1;

So we have to have to remember the variables that it has the post increment/decrement to evaluate after the expression is performed.

Assign statment (S *assign* E)

* Number assign=>[=,-=,\*=,/=,%=]
* Boolean assign=>[=,!=]
* String assign=>[=,+=]

Trivial cases

* E=function(arg1,arg2,arg3,…)[returns instanceof CObject]
* E=var[ instanceof CObject]
* E=string[\”everything\”]
* E=double [-max\_double, max\_double]
* E=bool [“true”, “false”]

Priorities

The expression can be evaluated in priority on our AST if you enclose it within parenthesis ( () ). In other the priority in given depending of operator. We establish these priorities.

=,new,(),., ++E,--E, \*, /,&&,+, -,||,<<,>>, ,

The pre post increment operators (i.e E++,E--) have to be remembered after the current expression is execution. Example,

++E1\*E2+E3

First time the ++E1 will be executed then the E1\*E2 and finally the E3.

The () and functions go through the most deepest to less,

Example,

(expresion1\*(expresion2\*(expression3))

First, expression3 will be processed then the expression2 and finally the expression1.

Example,

So after deciding the priorities, for example if we want to evaluate the following expression,

var s=4+5+6\*8+(8+10)\*10;

var s=4+5+6\*8+18\*10;

var s=4+5+48+180;

var s=9+228;

var s=237;

It can be deduced as expressions as,

var s=4+5+6\*8+(8+10)\*10;

var s=4+5+6\*8+(E0+E1)\*10;

E2 E3

var s=4+5+6\*8+E2\*E3;

E7 E8E5 E6 E4

var s=E7+E8+E5\*E6+E4;

E10 E9

var s=E10+E9+E4;

E11

var s=E11+E4

E12

var s=E12 (trivial case)

And the resulted AST will be,

As we can see at the picture, each expression in AST is expressed as node. A node can be an expression itself or trivial identification as variable or number in its leafs (terminal nodes). The most priority operators are more deeper against non priority operators.

With the example of X.XX can be deduce our process to construct our AST.

1. Detect valid operable expression (ends with ;)
2. Detect parenthesis and build AST for them first in recursive manner. Take the within parenthesis and pass it to the function.
3. If there’s not parenthesis in string, split it into tokens and expressions.
4. Take the most priority operators and simplify them as sub expressions.

Porting AST to C,

Typedef struct{

String expression; // variable or value.

String operator; // type of operator.

Vector<PAST\_Node> args; // in case is function.

PAST\_Node left, right;

} tAST\_Node;

Loop/Conditional expressions

The “while” has this pattern,

**While**(booleanStatment){

// sequence statements (inner)

}

Or ,

**While**(booleanStatment)

// sequence statement

The while can be evaluated as this flux diagram,

booleanStatment ==true?

no

yes

The “for” has this pattern,

// sequence statement/s

**for**(SequenceStatmentBeforeStart; booleanStatment; SequenceStatmentAfterProcess){

// Sequence Statments (inner)

}

Or

**for**(SequenceStatmentBeforeStart; booleanStatment; SequenceStatmentAfterProcess)

// SequenceStatment;

The while can be evaluated as this flux diagram,

The “do-while” has this pattern,

yes

no

SequenceStatmentBeforeStart

SequenceStatments

SequenceStatmentAfterProcess

booleanStatment ==true?

**do**{

// sequence statement (inner)

}**while**(BooleanExpression);

Or

**do**

// sequence statement (inner)

**while**(BooleanExpression);

the do-while can be evaluated

Execute expression/s

no

BooleanExpression==true?

yes

The “If” has this pattern,

**If**(BooleanExpression){

// Expressions

}[**elif**(BooleanExpression){ \*

// Expressions

}**else**{

// Expressions

}]

Or expressed

**If**(BooleanExpression)

Expression (NoVoid);

[**Elif**(BooleanExpression)

Expression (NoVoid);

**Else**

Expression (NoVoid);]

the if-else can be evaluated as,

yes

no

BooleanExpression==true?

Execute expression/s

**Else or elif**

Execute expression/s

Switch statement,

**switch**(value){

**case** value1: // the same as if(value==value1)

// Expressions1;

**break**;

**case** value2: // the same as elif(value==value2)

// Expressions2;

**break**;

…

**default**: // the as else

// Expressions\_default;

**break**;

}

The switch statement can be deduced from if-else,

if(value==value1){

// Expressions1;

}elif(value==value2){

// Expressions2;

}

…

}else{

// Expression\_else;

}

With the examples of we can be deduce our process to construct our AST.

1. Detect a if-else or while-for loop (if,for, do, while)
2. In function each type of element, deduce expressions.

Functions

Our script engine, will support functions. A function is a jump at some point of memory address and before ends, the function call put return value or not.

The function has this format,

**function** fun\_name(arg1, arg2, arg3,…){

// Expressions.

// return value or variable.

}

Before jump to the function, the cpu has to psuh the arguments through a stack. When the program jump to the function, then the cpu has to pop arguments from stack. We can treat the function as a scope but insead create a void stack is pre stacked with the arguments.

The returning argument it can be some cpu register or returning var register.

As another example that involves functions and its arguments with preincrement variables,

4+fun1(++i\*7,9\*fun2(j++)+5,10,fun3(++k,fun4(1,2,3,4,++l)+1))\*(8+5)

In function of operators priority, we start to find the most deeper function call in our case fun4, and then make operator,

4+fun1(++i\*7,9\*fun2(j++)+5,10,fun3(++k,fun4(1,2,3,4,++l)+1))\*(8 + 5)

E0 E2 E1 E2

4+fun1(E0\*7,9\*fun2(j++)+5,10,fun3(E1,fun4(1 ,2 ,3 ,4, E2)+1))\*(E1+E2)

E3 E4 E5 E6

4+fun1(E0\*7,9\*fun2(j++)+5,10,fun3(E1,fun4(E3,E4,E5,E6,E2)+1))\*(8+5)

E7

4+fun1(E0\*7,9\*fun2(j++)+5,10,fun3(E1,E7,+1))\*(8+5)

Classes

Our script will support classes. A class is defined as C++ as,

**Class** CMyCustomClass {

**var** i; // undefined member var

**var** j;// undefined member var

**function** my\_class(){ // constructor

i=0; // now I is integer

j=”striing”; // j is a string

}

};

var g=new CMyCustomClass(); // instance object type CMyCustomClass.

Classes can treated as programs because it has its global variables (called member vars) and its member functions. Because member vars and functions can be conflicted with global vars, we introduce the operator this to make sure the variable is referred to its member.

this.i=0;

Once all possible operators are defined, we are going to define the expression grammar,

The member vars are stacked in the new object you creates in script, but the functions are located in somewhere in the code. Then, when a member function is called,you makes the normal as normal function but you stack the object itself.

**Class** CMyCustomClassDerived {

**var** i; // undefined member var

**var** j;// undefined member var

**function** my\_class(){ // constructor

i=0; // now I is integer

j=”striing”; // j is a string

}

function fun1(){

return 0;

}

};

**Class** CMyCustomClassDerived extend CMyCustomClass {

**var** i; // undefined member var

**var** j;// undefined member var

**function** my\_class(){ // constructor

i=0; // now I is integer

j=”striing”; // j is a string

}

function fun1(){

super.fun1(); // or not!

}

};

The upper functions call be called using *super* operator

Basic expressions can be deduced but in others objects have to have convertions functions (i.e toNumber(), toString(), toBoolean()) to tell the compiler the type before perform operators +,-, etc for each basic types.

Buiding a parser

This is may be the hard step to do. Because we need to build a partse...

Tokenize

The first part of the parser is split all text in tokens to let have a more simple way to go through the main parts and evaluate whether an expression is wrong or not.

For example,

I we have the tokens +,\* and / we can express this expression,

(5+7)\*5+8\*9+(8+7+10)/100+50;

In a three like that,