//Code Gen File

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

#include "codegen.h"

#include <string>

using namespace std;

size\_t cg::Emit1(const unsigned char c)

{

size\_t addr = \_currentAddr;

buffer[\_currentAddr++] = c;

return addr;

}

size\_t cg::Emit2(unsigned short c)

{

size\_t addr = CurrentAddr();

unsigned char\* d = (unsigned char\*)&c;

buffer[\_currentAddr++] =d[0];

buffer[\_currentAddr++] =d[1];

return addr;

}

size\_t cg::Emit4(unsigned int c)

{

size\_t addr = CurrentAddr();

cg::Emit2(c & 0xFFFF);

cg::Emit2(c & 0xFFFF0000 );

return addr;

}

void cg::flush()

{

out->clear();

out->write(buffer, CurrentAddr());

out->flush();

}

void cg::BackPatch1(size\_t line, unsigned char value)

{

buffer[line] = value;

return;

}

void cg::BackPatch2(size\_t line, unsigned short value)

{

buffer[line++] = (value >> 0) & 0xFF;

buffer[line++] = (value >> 8) & 0xFF;

return;

}

void cg::BackPatch4(size\_t line, unsigned int value)

{

buffer[line++] = (value & 0xFF);

buffer[line++] = (value & 0xFF00);

buffer[line++] = (value & 0xFF000);

buffer[line++] = (value & 0xFF000);

return;

}

void cg::Log(std::string s)

{

if (PRINT)

cout << s;

}

size\_t cg::CurrentAddr()

{

return cg::\_currentAddr;

}

//Code Gen File Header

#ifndef CODEGEN\_H

#define CODEGEN\_H

#include <iostream>

#define PRINT true

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

class cg {

private:

char buffer[256 \*256];

size\_t \_currentAddr;

std::ostream\* out;

public:

cg(std::ostream& out) { this->out = &out; \_currentAddr = 0;}

void flush();

**//Emits the instruction/operand to the output stream**

**//Returns the line which was written to.**

size\_t Emit1(const unsigned char c);

size\_t Emit2(const unsigned short c);

size\_t Emit4(const unsigned int c);

**//outputs syntax to screen, if the tree is wanted to be viewed.**

void Log(std::string);

size\_t CurrentAddr();

void BackPatch1(size\_t line, unsigned char value);

void BackPatch2(size\_t line, unsigned short value);

void BackPatch4(size\_t line, unsigned int value);

};

#endif

//doggieTypes

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

#include <iostream>

#include "doggieTypes.h"

std::ostream& operator<<(std::ostream& os, enum Type t)

{

if (t == TypeYorkie)

return os << "Yorkie";

if (t == TypeDalmatian)

return os << "Dalmatian";

if (t == TypePoodle)

return os << "Poodle";

if (t == TypeCorgi)

return os << "Corgi";

if (t == TypeDachshund)

return os << "Dachshund";

if (t == TypeStray)

return os << "Stray";

return os << "\_UNKOWN\_TYPE\_";

}

std::ostream& operator<<(std::ostream& os, enum Kind k)

{

if (k == KindVar)

return os << "var";

else if (k == KindFunc)

return os << "func";

return os << "\_UNKOWN\_KIND\_";

}

//Doggietypes header

#ifndef DOGGIETYPES\_H

#define DOGGIETYPES\_H

**//Group Symbol Table Project**

// Jonathan Ringer

// Jeremy Vasseur

// Washing Kaleb

// Phillip Germagliotti

#include <iostream>

enum Type { TypeYorkie = 0, TypeDalmatian, TypePoodle, TypeCorgi, TypeDachshund, TypeStray};

//not sure about the void type, depends on the size of a word in memory

const int typeSize[] = { 1, 1, 4, 4, 4, 2, 2};

enum Kind { KindVar = 0, KindFunc };

inline int TypeSize(Type type) { return typeSize[type]; }

std::ostream& operator<<(std::ostream& os, enum Type t);

std::ostream& operator<<(std::ostream& os, enum Kind k);

#endif

//Interpreter

#ifndef INTERPRETER\_CPP

#define INTERPRETER\_CPP

/\* Kian Pokorny January 8, 2010

This file contains an interpreter for the x machine.

The input file must be in the same directory as the

interpreter. At the prompt the file name is typed in

including any extension.

Fix all code functions that have

\*/

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

#include <iostream>

#include <iomanip>

#include <cstdlib>

#include <cstdio>

#include <fstream>

#include <ctype.h>

#include <string>

using namespace std;

/\*\*\*\*\*\*\*\*\*\* Op-codes \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const unsigned char INC = 0x01;

const unsigned char DEC = 0x02;

const unsigned char ADD = 0x03;

const unsigned char SUB = 0x04;

const unsigned char MUL = 0x05;

const unsigned char DIV = 0x06;

const unsigned char MOD = 0x07;

const unsigned char NEG = 0x08;

const unsigned char ADDF = 0x11;

const unsigned char SUBF = 0x12;

const unsigned char MULF = 0x13;

const unsigned char DIVF = 0x14;

const unsigned char MODF = 0x15;

const unsigned char NEGF = 0x16;

const unsigned char NOT = 0x17;

const unsigned char AND = 0x18;

const unsigned char OR = 0x19;

const unsigned char XOR = 0x20;

const unsigned char SHR = 0x21;

const unsigned char SHL = 0x22;

const unsigned char SHRA = 0x23;

const unsigned char ROR = 0x25;

const unsigned char ROL = 0x26;

const unsigned char HALT = 0x27;

const unsigned char DER = 0x28;

const unsigned char TPS = 0x31;

const unsigned char TNG = 0x32;

const unsigned char TZE = 0x33;

const unsigned char TNP = 0x34;

const unsigned char TNN = 0x35;

const unsigned char TNZ = 0x36;

const unsigned char TPSF = 0x41;

const unsigned char TNGF = 0x42;

const unsigned char TZEF = 0x43;

const unsigned char TNPF = 0x44;

const unsigned char TNNF = 0x45;

const unsigned char TNZF = 0x46;

const unsigned char PUSL = 0x50;

const unsigned char PUSG = 0x51;

const unsigned char PUSP = 0x52;

const unsigned char PUSS = 0x53;

const unsigned char POPL = 0x54;

const unsigned char POPG = 0x55;

const unsigned char POPP = 0x56;

const unsigned char POPS = 0x57;

const unsigned char INS = 0x58;

const unsigned char OTS = 0x59;

const unsigned char BYES = 0x60;

const unsigned char BNO = 0x61;

const unsigned char JMP = 0x62;

const unsigned char JMPD = 0x63;

const unsigned char JMPI = 0x64;

const unsigned char JMPR = 0x65;

const unsigned char LDD1 = 0x70;

const unsigned char LDDN1 = 0x71;

const unsigned char LDI1 = 0x72;

const unsigned char LDIN1 = 0x73;

const unsigned char LDR1 = 0x74;

const unsigned char LDRN1 = 0x75;

const unsigned char LDM1 = 0x76;

const unsigned char LDLR1 = 0x79;

const unsigned char LDLN1 = 0x80;

const unsigned char LDGR1 = 0x81;

const unsigned char LDGN1 = 0x82;

const unsigned char STD1 = 0x83;

const unsigned char STDN1 = 0x84;

const unsigned char STI1 = 0x85;

const unsigned char STIN1 = 0x86;

const unsigned char STR1 = 0x87;

const unsigned char STRN1 = 0x88;

const unsigned char STLR1 = 0x92;

const unsigned char STLN1 = 0x93;

const unsigned char STGR1 = 0x94;

const unsigned char STGN1 = 0x95;

const unsigned char LDD2 = 0xA0;

const unsigned char LDDN2 = 0xA1;

const unsigned char LDI2 = 0xA2;

const unsigned char LDIN2 = 0xA3;

const unsigned char LDR2 = 0xA4;

const unsigned char LDRN2 = 0xA5;

const unsigned char LDM2 = 0xA6;

const unsigned char LDLR2 = 0xA7;

const unsigned char LDLN2 = 0xB0;

const unsigned char LDGR2 = 0xB1;

const unsigned char LDGN2 = 0xB2;

const unsigned char STD2 = 0xB3;

const unsigned char STDN2 = 0xB4;

const unsigned char STI2 = 0xB5;

const unsigned char STIN2 = 0xB6;

const unsigned char STR2 = 0xB7;

const unsigned char STRN2 = 0xB8;

const unsigned char STLR2 = 0xC2;

const unsigned char STLN2 = 0xC3;

const unsigned char STGR2 = 0xC4;

const unsigned char STGN2 = 0xC5;

const unsigned char LDD4 = 0xD0;

const unsigned char LDDN4 = 0xD1;

const unsigned char LDI4 = 0xD2;

const unsigned char LDIN4 = 0xD3;

const unsigned char LDR4 = 0xD4;

const unsigned char LDRN4 = 0xD5;

const unsigned char LDM4 = 0xD6;

const unsigned char LDLR4 = 0xD9;

const unsigned char LDLN4 = 0xE0;

const unsigned char LDGR4 = 0xE1;

const unsigned char LDGN4 = 0xE2;

const unsigned char STD4 = 0xE3;

const unsigned char STDN4 = 0xE4;

const unsigned char STI4 = 0xE5;

const unsigned char STIN4 = 0xE6;

const unsigned char STR4 = 0xE7;

const unsigned char STRN4 = 0xE8;

const unsigned char STLR4 = 0xF2;

const unsigned char STLN4 = 0xF3;

const unsigned char STGR4 = 0xF4;

const unsigned char STGN4 = 0xF5;

const unsigned char NOOP = 0xFF;

const unsigned char DSTK = 0xF6;

const unsigned char DROP = 0x47;

const unsigned char FLT = 0x48;

const unsigned char SWAP = 0x49;

const unsigned char OUTC = 0x4A;

const unsigned char OUTI = 0x4B;

const unsigned char OUTR = 0x4C;

const unsigned char COPY = 0x1F;

const unsigned char DSTCK = 0x1B;

const unsigned char INPUTC = 0x5A;

const unsigned char INPUTI = 0x5B;

const unsigned char INPUTR = 0x5C;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* machine \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool done;

bool quit;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* config \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const unsigned int MemSize = 256 \* 256; /\* 64K needs to be a power of 2\*/

const unsigned int StackSize = 1024;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* convert \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

float real(unsigned int x) {

int i = x;

float \*f;

f = (float \*) &i;

return \*f;

}

unsigned int unreal(float x) {

float f = x;

int \*i;

i = (int \*) &f;

return \*i;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Registers \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

unsigned int PC; /\* Program Counter\*/

unsigned char IR; /\* Instruction Register\*/

unsigned int T; /\* Top of Expression Stack\*/

/\* Special Purpose Registers\*/

unsigned int S;

unsigned int L;

unsigned int G;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* EXPRESSION Stack \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

unsigned int X[StackSize] = {0}; /\*\*\* Expression Stack\*/

unsigned int pop(void);

void push(unsigned int value);

void push(unsigned short value);

void push(unsigned char value);

void DumpStack();

unsigned int pop(void){

int value;

if(T){

value = X[--T];

} else {

cout << "Expressions Stack Underflow. on line " << PC << endl;

done = true;

}

return value;

}

void push( unsigned char value)

{unsigned int x;

if (0x80 & value) x = value | 0xffffff00;

else x = value;

push(x);

}

void push(unsigned short value)

{unsigned int x;

if ( 0x8000 & value) x = value | 0xffff0000;

else x = value;

push(x);

}

void push(unsigned int value) {

if (T < StackSize) {

X[T++] = value;

} else {

cout << "Expression Stack Overflow." << endl;

/\*done = true; \*/

}

}

void DumpStack()

{cout << endl << "Dumping Stack" ;

int i;

char tach;

for ( i = (T-1); i >=0; i--) cout << endl << X[T];

cout << " End stack " << endl;

cin >> tach;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Memory \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const unsigned int AddrMask = MemSize - 1;

unsigned char M[MemSize] = {0}; /\* RAM\*/

unsigned int get4(unsigned int & addr);

unsigned int get2(unsigned int & addr);

unsigned int get1(unsigned int & addr);

void put1(unsigned int & addr, unsigned int value);

void put2(unsigned int & addr, unsigned int value);

void put4(unsigned int & addr, unsigned int value);

unsigned int get4(unsigned int & addr) {

int a = get2(addr);

int b = get2(addr) << 16;

return a | b;

}

unsigned int get2(unsigned int & addr) {

int a = get1(addr);

int b = get1(addr) << 8;

return a | b;

}

unsigned int get1(unsigned int & addr) {

return M[addr++ & AddrMask];

}

void put4(unsigned int & addr, unsigned int value) {

put2(addr,value);

put2(addr,value >> 16);

}

void put2(unsigned int & addr, unsigned int value) {

put1(addr,value);

put1(addr,value >> 8);

}

void put1(unsigned int & addr, unsigned int value) {

M[addr++ & AddrMask] = value & 0xFF;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* loader \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const int MaxCommandLineLen = 80;

/\* Made Global so that xdecode can write same filename.\*/

char commandLine[ MaxCommandLineLen];

void load( char prompt = '>');

void load( char prompt) {

bool loaded = false;

char fname[ 80];

cout << endl;

while( !loaded) {

cout << prompt;

//cin.getline( fname, MaxCommandLineLen);

/\* strcpy( fname, commandLine);

strcat( fname, ".x");

\*/ fstream fin("sample.x");

ofstream fout("debug.txt", ios::out);

if( !fin) {

cout << "\*\*\* ERROR: Invalid Filename" << endl;

} else {

fin.read( reinterpret\_cast< char \*>(&M), MemSize);

/\* if( !fin) {

cout << "\*\*\* ERROR: File Error" << endl;

} else {

\*/ loaded = true;

/\* } \*/

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Fetch \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Routines for fetching the next instruction and stored operands\*/

unsigned int next4(void);

unsigned int next2(void);

unsigned int next1(void);

unsigned int next4(void){

int a = next2();

int b = next2() << 16;

return a | b ;

}

unsigned int next2(void) {

int a = next1();

int b = next1() << 8;

return a | b;

}

unsigned int next1(void) {

return get1(PC);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main() {

int IR = 0;

const unsigned int MaxInputLen = 80;

unsigned int effAddr;

unsigned int a,b,c,d;

unsigned int n;

unsigned int i;

char ch;

float r;

char k[ MaxInputLen];

/\* ofstream pcdata ("pcdata.dat",ios::out);

\*/

quit = false;

/\* ===============>Test codes

//for(n=0;n<512;n++)

// X[n]=rand();

\*/

FILE \*ifp;

ifp=fopen("test.out","w");

ofstream testdata("testdata.txt",ios::out);

/\*<===============\*/

while (!quit) {

load(0xAF); /\* 0xAF is '»', giving user a prompt to enter input filename. \*/

T = 0;

done = false;

PC = 0;

S = 1;

L = 1;

G = 1;

while (!done) {

/\*===============>Test codes\*/

testdata<<"List of Registers:"<<endl;

testdata<<"PC:"<<PC;

/\* testdata<<"\tIR:"<<setbase(10)<<IR;

\*/ testdata<<"\tT:"<<T;

testdata<<"\tS:"<<S;

testdata<<"\tL:"<<L;

testdata<<"\tG:"<<G<<endl;

testdata<<"List of Stack:"<<endl;

for (n=0;n<T;n++){

testdata<< X[n]<<"\t";

if((n+1)%10==0) testdata<<endl;

}

fprintf(ifp,"List of Registers:\n");

fprintf(ifp,"PC: %d",PC);

fprintf(ifp,"\tIR: %d",IR);

fprintf(ifp,"\tT: %d",T);

fprintf(ifp,"\tS: %d",S);

fprintf(ifp,"\tL: %d",L);

fprintf(ifp,"\tG: %d",G);

fprintf(ifp,"List of Stack: \n");

for (n=0;n<T;n++){

fprintf(ifp," %d \t ",X[n]);

if((n+1)%10==0) fprintf(ifp," \n");

}

fprintf(ifp,"\n------------------------------------------------------\n");

testdata<<endl<<"------------------------------------------------------"<<endl;

cout.setf(ios::dec);

for( n=0;n<48;n++){

fprintf(ifp," %d \t ",M[G+n]);

if((n+1)%10==0) fprintf(ifp," \n");

}

fprintf(ifp,"\n=======================================================\n");

testdata<<endl<<"======================================================="<<endl;

/\*<===============\*/

IR = next1(); /\* fetch\*/

/\* cout << IR << endl;\*/

/\*

// pcdata << "PC: " << hex << PC-1 << " M[17]: " << M[23] << dec << endl;

cout << "PC: " << hex << PC << " M[17]: " << M[23] << dec << endl;

\*/

switch( IR) {

**/\* Integer Arithmetic Operations\*/**

case INC :

push (pop() + 1);

break;

case DEC :

push (pop() - 1);

break;

case ADD :

b = pop();

push(pop() + b);

break;

case SUB :

b = pop();

push(pop() - b);

break;

case MUL :

b = pop();

push(pop() \* b);

break;

case DIV :

b = pop();

push(pop() / b);

break;

case MOD :

b = pop();

push(pop() % b);

break;

case NEG :

push(pop() \* -1);

break;

/\* Floating-point Arithmetic Operations\*/

case ADDF : /\*Floating-point Add\*/

b = pop();

push( unreal( real( pop()) + real( b)));

break;

case SUBF : /\*Floating-point Subtraction\*/

b = pop();

push(unreal(real(pop()) - real(b)));

case MULF : /\*Floating-point Multiplcation\*/

b = pop();

push( unreal( real( pop())\*real( b)));

break;

case DIVF : /\*Floating-point Division\*/

b = pop();

push( unreal( real( pop()) / real( b)));

break;

case NEGF : /\*Floating-point Negation\*/

push( unreal( real( pop()) \* -1));

break;

/\* Bitwise Logical Operations\*/

case NOT :

push( ~pop());

break;

case AND :

b = pop();

push( pop() & b);

break;

case OR :

b = pop();

push(pop() | b);

break;

case XOR :

b = pop();

push(pop() ^ b);

break;

/\* Shift Operations\*/

case SHR :

push(( pop() >> 1) & 0x7FFFFFFF);

break;

case SHL :

push((pop() << 1) & 0xFFFFFFFE);

break;

case SHRA : /\* Arithmetic Shift Right\*/

push(pop() >> 1);

break;

case ROR : /\* Rotate Right\*/

a = pop();

b = ( a & 0x00000001) << 31;

push((( a >> 1) & 0x7FFFFFFF) | b);

break;

case ROL : /\* Rotate Left\*/

a = pop();

b = ( a & 0x80000000) >> 31;

push((( a << 1) & 0xFFFFFFFE) | b);

break;

/\* Test Integer Operations\*/

case TPS :

n = pop();

if(n > 0) push(0xFFFFFFFF); else push((unsigned int)0);

break;

case TNG :

n = pop();

if(n < 0) push(0xFFFFFFFF); else push((unsigned int)0);

break;

case TZE :

n = pop();

if(n == 0) push(0xFFFFFFFF); else push((unsigned int)0);

break;

case TNP : //Test not positive

if (n <= 0) push(0xFFFFFFFF); else push((unsigned int)0);

break;

case TNN :

n = pop();

if(n >= 0) push(0xFFFFFFFF); else push((unsigned int)0);

break;

case TNZ :

n = pop();

if(n != 0) push(0xFFFFFFFF); else push((unsigned int)0);

break;

**/\* Test Floating-point Operations\*/**

case TPSF : /\* Test Positive (floating-point)\*/

if (real(pop()) > 0.0) push(0xFFFFFFFF); else push((unsigned int)0);

break;

case TNGF : /\* Test Negative (floating-point)\*/

if (real(pop()) < 0.0) push(0xFFFFFFFF); else push((unsigned int)0);

break;

case TZEF : /\* Test Zero (floating-point)\*/

if (real(pop()) == 0.0) push(0xFFFFFFFF); else push((unsigned int)0);

break;

case TNPF : /\* Test Not Positive (floating-point)\*/

if (real(pop()) <= 0.0) push(0xFFFFFFFF); else push((unsigned int)0);

break;

case TNNF : /\* Test Not Negative (floating-point)\*/

if (real(pop()) >= 0) push(0xFFFFFFFF); else push((unsigned int)0);

break;

case TNZF : /\* Test Not Zero (floating-point)\*/

if (real(pop()) != 0.0) push(0xFFFFFFFF); else push((unsigned int)0);

break;

/\* Register Transfer Operations\*/

case PUSL : /\* Push L Onto Stack\*/

push (L);

break;

case PUSG : /\* Push G Onto Stack\*/

push (G);

break;

case PUSP : /\* Push PC Onto Stack\*/

push (PC);

break;

case PUSS : /\* Push S Onto Stack\*/

push (S);

break;

case POPL : /\* Pop L from the Stack\*/

L = pop();

break;

case POPG : /\* Pop G from the Stack\*/

G = pop();

break;

case POPP : /\* Pop PC from the Stack\*/

PC = pop();

break;

case POPS : /\* Pop S from the Stack\*/

S = pop();

break;

/\* IO Operations.\*/

case INS : /\* Input String from keyboard\*/

effAddr = pop();

cin.getline( k, MaxInputLen);

i = 0;

do {

if (k[i] == '\0') break;

else i++;

} while (i < MaxInputLen);

b = effAddr; /\* b starts as base address of str\*/

put1(b, i);

/\* // d = get1( b); // d is declared len as str

// c = strlen( k); // c is current 1 en of str

// if ( c > d) c = d; //the strings are safe

// M[ b++ & AddrMask] = c;

\*/

for( a = 0; a < i; a++) put1( b, k[ a]);

break;

case OTS : /\* Output String to Screen\*/

effAddr = pop();

b = effAddr;

c = get1( b); /\* c is cur len of str\*/

for( a = 0; a < c; a++) cout << ( char) ( get1( b) & 0xFF);

break;

/\* Branch Instructions\*/

case BYES : /\* Branch on Not Zero (True)\*/

a = pop();

b = next1();

if( a) PC = PC + b;

break;

case BNO : /\* Branch on Zero (False)\*/

a = pop();

b = next1();

if( !a) PC = PC + b;

break;

/\* Jump Instructions\*/

case JMP : /\* Jump (Unconditional) Implied\*/

PC = pop();

break;

case JMPD : /\* Jump (Unconditional) Direct\*/

a = next2();

PC = a;

break;

case JMPI : /\* Jump (Unconditional) Indirect\*/

a = next2();

PC = get2( a);

break;

case JMPR : /\* Jump (Unconditional) Relative\*/

a = next2();

PC += a;

break;

/\* Load Instructions\*/

case LDD1 : /\* Load Direct\*/

a = next2();

push(get1(a));

break;

case LDDN1 : /\* Load Direct Indexed\*/

a = next2() + pop();

push(get1(a));

break;

case LDI1 : /\* Load Indirect\*/

b = next2();

a = get2(b);

push(get1(a));

break;

case LDIN1 : /\* Load Indirect Indexed\*/

b = next2();

a = get2(b) + pop();

push(get1(a));

break;

case LDR1 : /\* Load Relative\*/

c = next1();

a = PC + c;

push(get1(a));

break;

case LDRN1 : /\* Load Relative Indexed\*/

c = next1();

a = PC + c + pop();

push(get1(a));

break;

case LDM1 : /\* Load Immediate 1 byte\*/

d = next1();

push(d);

break;

case LDLR1 : /\* Load Local Relative\*/

a = next2() + L;

push(get1(a));

break;

case LDLN1 : /\* Load Local Indexed\*/

a = next2() + pop() + L;

push(get1(a));

break;

case LDGR1 : /\* Local Global Relative\*/

a = next2() + G;

push(get1(a));

break;

case LDGN1 : /\* Local Global Indexed\*/

a = next2() + pop() + G;

push(get1(a));

break;

case LDD2 : /\* Load Direct\*/

a = next2();

push(get2(a));

break;

case LDDN2 : /\* Load Direct Indexed\*/

a = next2() + pop();

push(get2(a));

break;

case LDI2 : /\* Load Indirect\*/

b = next2();

a = get2(b);

push(get2(a));

break;

case LDIN2 : /\* Load Indirect Indexed\*/

b = next2();

a = get2(b) + pop();

push(get2(a));

break;

case LDR2 : /\* Load Relative\*/

c = next1();

a = PC + c;

push(get2(a));

break;

case LDRN2 : /\* Load Relative Indexed\*/

c = next1();

a = PC + c + pop();

push(get2(a));

break;

case LDM2 : /\* Load Immediate 2 byte\*/

d = next2();

push(d);

break;

case LDLR2 : /\* Load Local Relative\*/

a = next2() + L;

push(get2(a));

break;

case LDLN2 : /\* Load Local Indexed\*/

a = next2() + pop() + L;

push(get2(a));

break;

case LDGR2 : /\*/ Local Global Relative\*/

a = next2() + G;

push(get2(a));

break;

case LDGN2 : /\* Local Global Indexed\*/

a = next2() + pop() + G;

push(get2(a));

break;

case LDD4 : /\* Load Direct\*/

a = next2();

push(get4(a));

break;

case LDDN4 : /\* Load Direct Indexed\*/

a = next2() + pop();

push(get4(a));

break;

case LDI4 : /\* Load Indirect\*/

b = next2();

a = get2(b);

push(get4(a));

break;

case LDIN4 : /\* Load Indirect Indexed\*/

b = next2();

a = get2(b) + pop();

push(get4(a));

break;

case LDR4 : /\* Load Relative\*/

c = next1();

a = PC + c;

push(get4(a));

break;

case LDRN4 : /\* Load Relative Indexed\*/

c = next1();

a = PC + c + pop();

push(get4(a));

break;

case LDM4 : /\* Load Immediate 4 byte\*/

d = next4();

push(d);

break;

case LDLR4 : /\* Load Local Relative\*/

a = next2() + L;

push(get4(a));

break;

case LDLN4 : /\* Load Local Indexed\*/

a = next2() + pop() + L;

push(get4(a));

break;

case LDGR4 : /\* Local Global Relative\*/

a = next2() + G;

push(get4(a));

break;

case LDGN4 : /\* Local Global Indexed\*/

a = next2() + pop() + G;

push(get4(a));

break;

case STD1 : /\* Store Direct\*/

a = next2();

put1(a,pop());

break;

case STDN1 : /\* Store Direct Indexed\*/

a = next2() + pop();

put1(a,pop());

break;

case STI1 : /\* Store Indirect\*/

b = next2();

a = get2(b);

put1(a,pop());

break;

case STIN1 : /\* Store Indirect Indexed\*/

b = next2();

a = get2(b) + pop();

put1(a,pop());

break;

case STR1 : /\* Store Relative\*/

c = next1();

a = PC + c;

put1(a,pop());

break;

case STRN1 : /\* Store Relative Indexed\*/

c = next1();

a = PC + c + pop();

put1(a,pop());

break;

case STLR1 : /\* Store Local Relative\*/

a = next2() + L;

put1(a,pop());

break;

case STLN1 : /\* Store Local Indexed\*/

a = next2() + pop() + L;

put1(a,pop());

break;

case STGR1 : /\* Store Global Relative\*/

a = next2() + G;

put1(a,pop());

break;

case STGN1 : /\* Store Global Indeed\*/

a = next2() + pop() + G;

put1(a,pop());

break;

case STD2 : /\* Store Direct\*/

a = next2();

put2(a,pop());

break;

case STDN2 : /\* Store Direct Indexed\*/

a = next2() + pop();

put2(a,pop());

break;

case STI2 : /\* Store Indirect\*/

b = next2();

a = get2(b);

put2(a,pop());

break;

case STIN2 : /\* Store Indirect Indexed\*/

b = next2();

a = get2(b);

put2(a,pop());

break;

case STR2 : /\* Store Relative\*/

c = next1();

a = PC + c;

put2(a,pop());

break;

case STRN2 : /\* Store Relative Indexed\*/

c = next1();

a = PC + c + pop();

put2(a,pop());

break;

case STLR2 : /\* Store Local Relative\*/

a = next2() + L;

put2(a,pop());

break;

case STLN2 : /\* Store Local Indexed\*/

a = next2() + pop() + L;

put2(a,pop());

break;

case STGR2 : /\* Store Global Relative\*/

a = next2() + G;

put2(a,pop());

break;

case STGN2 : /\* Store Global Indeed\*/

a = next2() + pop() + G;

put2(a,pop());

break;

case STD4 : /\* Store Direct\*/

a = next2();

put4(a,pop());

break;

case STDN4 : /\* Store Direct Indexed\*/

a = next2() + pop();

put4(a,pop());

break;

case STI4 : /\* Store Indirect\*/

b = next2();

a = get2(b);

put4(a,pop());

break;

case STIN4 : /\* Store Indirect Indexed\*/

b = next2();

a = get2(b) + pop();

put4(a,pop());

break;

case STR4 : /\* Store Relative\*/

c = next1();

a = PC + c;

put4(a,pop());

break;

case STRN4 : /\* Store Relative Indexed\*/

c = next1();

a = PC + c + pop();

put4(a,pop());

break;

case STLR4 : /\* Store Local Relative\*/

a = next2() + L;

put4(a,pop());

break;

case STLN4 : /\* Store Local Indexed\*/

a = next2() + pop() + L;

put4(a,pop());

break;

case STGR4 : /\* Store Global Relative\*/

a = next2() + G;

put4(a,pop());

break;

case STGN4 : /\* Store Global Indexed\*/

a = next2() + pop() + G;

put4(a,pop());

break;

case DER : /\* Dereference\*/

a = pop();

push(get2(a));

break;

case HALT : /\* Halt the processor\*/

done = true;

break;

case FLT : /\* Float an Int\*/

d = pop();

r = d;

push( unreal( r));

break;

case COPY :

b = pop();

push(b);

push(b);

break;

case DROP : /\* Drop Top of Stack\*/

pop();

break;

case SWAP : /\* Swap Top Two\*/

a = pop();

b = pop();

push( a);

push( b);

break;

case OUTC : /\* Output a Char\*/

ch = pop() & 0xff;

cout << ch;

break;

case OUTI : /\* Output a Int\*/

n = pop();

cout << n;

break;

case OUTR : /\* Output a Int\*/

d = pop();

r = real( d);

cout << r;

break;

case INPUTC : /\*Input a Char\*/

cin >> ch;

push( (unsigned char)(ch));

break;

case INPUTI : /\*Input an Int\*/

cin >> n;

push( (unsigned int)(n));

break;

case INPUTR : /\*Input a real\*/

cin >> r;

push( unreal(r));

break;

/\* woohoo, added this one fer m'own purpose\*/

case DSTK: /\* Grab top stack val and shove it to the screen\*/

{cout << "Stack: ";

cout << hex;

for (size\_t i = 0; i < T; i++) cout << X[i] << " ";

cout << dec << endl;

break;

}

case NOOP : break; /\* No operation\*/

default : /\* No instruction match up with op code\*/

cout << "Invalid Instruction. PC:" << hex << PC << dec << endl;

done = true;

break;

}/\* End switch\*/

} /\*End while(!done)\*/

} /\* End while(!quit) \*/

testdata.close();

/\* close(ifp);\*/

} /\*\* End Main \*/

#endif

//Interpreter Header

#ifndef INTERPRETER\_H

#define INTERPRETER\_H

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

namespace opcode {

const unsigned char INC = 0x01;

const unsigned char DEC = 0x02;

const unsigned char ADD = 0x03;

const unsigned char SUB = 0x04;

const unsigned char MUL = 0x05;

const unsigned char DIV = 0x06;

const unsigned char MOD = 0x07;

const unsigned char NEG = 0x08;

const unsigned char ADDF = 0x11;

const unsigned char SUBF = 0x12;

const unsigned char MULF = 0x13;

const unsigned char DIVF = 0x14;

const unsigned char MODF = 0x15;

const unsigned char NEGF = 0x16;

const unsigned char NOT = 0x17;

const unsigned char AND = 0x18;

const unsigned char OR = 0x19;

const unsigned char XOR = 0x20;

const unsigned char SHR = 0x21;

const unsigned char SHL = 0x22;

const unsigned char SHRA = 0x23;

const unsigned char ROR = 0x25;

const unsigned char ROL = 0x26;

const unsigned char HALT = 0x27;

const unsigned char DER = 0x28;

const unsigned char TPS = 0x31;

const unsigned char TNG = 0x32;

const unsigned char TZE = 0x33;

const unsigned char TNP = 0x34;

const unsigned char TNN = 0x35;

const unsigned char TNZ = 0x36;

const unsigned char TPSF = 0x41;

const unsigned char TNGF = 0x42;

const unsigned char TZEF = 0x43;

const unsigned char TNPF = 0x44;

const unsigned char TNNF = 0x45;

const unsigned char TNZF = 0x46;

const unsigned char PUSL = 0x50;

const unsigned char PUSG = 0x51;

const unsigned char PUSP = 0x52;

const unsigned char PUSS = 0x53;

const unsigned char POPL = 0x54;

const unsigned char POPG = 0x55;

const unsigned char POPP = 0x56;

const unsigned char POPS = 0x57;

const unsigned char INS = 0x58;

const unsigned char OTS = 0x59;

const unsigned char BYES = 0x60;

const unsigned char BNO = 0x61;

const unsigned char JMP = 0x62;

const unsigned char JMPD = 0x63;

const unsigned char JMPI = 0x64;

const unsigned char JMPR = 0x65;

const unsigned char LDD1 = 0x70;

const unsigned char LDDN1 = 0x71;

const unsigned char LDI1 = 0x72;

const unsigned char LDIN1 = 0x73;

const unsigned char LDR1 = 0x74;

const unsigned char LDRN1 = 0x75;

const unsigned char LDM1 = 0x76;

const unsigned char LDLR1 = 0x79;

const unsigned char LDLN1 = 0x80;

const unsigned char LDGR1 = 0x81;

const unsigned char LDGN1 = 0x82;

const unsigned char STD1 = 0x83;

const unsigned char STDN1 = 0x84;

const unsigned char STI1 = 0x85;

const unsigned char STIN1 = 0x86;

const unsigned char STR1 = 0x87;

const unsigned char STRN1 = 0x88;

const unsigned char STLR1 = 0x92;

const unsigned char STLN1 = 0x93;

const unsigned char STGR1 = 0x94;

const unsigned char STGN1 = 0x95;

const unsigned char LDD2 = 0xA0;

const unsigned char LDDN2 = 0xA1;

const unsigned char LDI2 = 0xA2;

const unsigned char LDIN2 = 0xA3;

const unsigned char LDR2 = 0xA4;

const unsigned char LDRN2 = 0xA5;

const unsigned char LDM2 = 0xA6;

const unsigned char LDLR2 = 0xA7;

const unsigned char LDLN2 = 0xB0;

const unsigned char LDGR2 = 0xB1;

const unsigned char LDGN2 = 0xB2;

const unsigned char STD2 = 0xB3;

const unsigned char STDN2 = 0xB4;

const unsigned char STI2 = 0xB5;

const unsigned char STIN2 = 0xB6;

const unsigned char STR2 = 0xB7;

const unsigned char STRN2 = 0xB8;

const unsigned char STLR2 = 0xC2;

const unsigned char STLN2 = 0xC3;

const unsigned char STGR2 = 0xC4;

const unsigned char STGN2 = 0xC5;

const unsigned char LDD4 = 0xD0;

const unsigned char LDDN4 = 0xD1;

const unsigned char LDI4 = 0xD2;

const unsigned char LDIN4 = 0xD3;

const unsigned char LDR4 = 0xD4;

const unsigned char LDRN4 = 0xD5;

const unsigned char LDM4 = 0xD6;

const unsigned char LDLR4 = 0xD9;

const unsigned char LDLN4 = 0xE0;

const unsigned char LDGR4 = 0xE1;

const unsigned char LDGN4 = 0xE2;

const unsigned char STD4 = 0xE3;

const unsigned char STDN4 = 0xE4;

const unsigned char STI4 = 0xE5;

const unsigned char STIN4 = 0xE6;

const unsigned char STR4 = 0xE7;

const unsigned char STRN4 = 0xE8;

const unsigned char STLR4 = 0xF2;

const unsigned char STLN4 = 0xF3;

const unsigned char STGR4 = 0xF4;

const unsigned char STGN4 = 0xF5;

const unsigned char NOOP = 0xFF;

const unsigned char DSTK = 0xF6;

const unsigned char DROP = 0x47;

const unsigned char FLT = 0x48;

const unsigned char SWAP = 0x49;

const unsigned char OUTC = 0x4A;

const unsigned char OUTI = 0x4B;

const unsigned char OUTR = 0x4C;

const unsigned char COPY = 0x1F;

const unsigned char DSTCK = 0x1B;

const unsigned char INPUTC = 0x5A;

const unsigned char INPUTI = 0x5B;

const unsigned char INPUTR = 0x5C;

}

#endif

//Main

//Main file for doggie parser, essentially just a wrapper for the parser itself

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

#include <iostream>

#include "Parser.h"

#include "Interpreter.h"

#include <sstream>

#include <fstream>

#include "Parser.h"

using namespace std;

int main() {

fstream f("test.ds");

ofstream o("output.x");

if (f)

{

Parser p(&f, o);

p.Run();

p.Flush();

}

else {

cout << "Couldn't find file" << endl;

}

f.close();

o.close();

f.open("output.x");

//cout << f.rdbuf();

cout << endl;

cout << endl << "done parsing file." << endl;

cout << "hit any key to continue....." << endl;

char c;

cin >> c;

return 0;

}

//Parser

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

#include "Parser.h"

#include <iostream>

#include <sstream>

#include <string>

using namespace std;

using namespace opcode;

int to\_int(string &s)

{

int result;

stringstream ss;

ss << s;

ss >> result;

return result;

}

template<typename T>

std::string to\_string( const T &n)

{

stringstream ss;

ss << n;

return ss.str();

}

TokenInfo Parser::GetNextToken()

{

if (s.HasToken())

currentToken = s.NextToken();

return currentToken;

}

bool Parser::Yorkie(enum Type &t)

{

Expr0(t);

return false;

}

bool Parser::Corgi(enum Type &t)

{

Expr0(t);

return false;

}

bool Parser::Dalmatian(enum Type &t)

{

Expr0(t);

return false;

}

bool Parser::ArrayDeclaration(enum Type t)

{

if (Match(ArrayDefToken))

{

if (currentToken.token == IntToken)

{

int num = to\_int(currentToken.value);

GetNextToken();

if (Match(ArrayDefToken))

{

if (Match(EoLToken))

{

//TODO: add num records to symbol table

return true;

}

//TODO handle else's

}

}

}

return false;

}

bool Parser::ArrayLHS(enum Type t, std::string name)

{

if (Match(ArrayDefToken))

{

int arraySize = 0;

//arrays cannot be dynamically allocated on the stack

if (currentToken.token == IntToken)

{

int size = to\_int(currentToken.value);

table.AddArray(name, t, size);

if (Match(ArrayDefToken))

{

}

}

}

return false;

}

bool Parser::Poodle(int &value)

{

//TODO emit code

if(currentToken.token == IntToken) {

int size = to\_int(currentToken.value);

GetNextToken();

//TODO emit size

c.Emit1(LDM4);

c.Emit4(size);

return true;

}

else if (currentToken.token == NameToken)

{

Record r = table.FindRecord(currentToken.value);

if(r.type == TypePoodle) {

//TODO emit code for finding record at specific scope

c.Emit1(LDM2);

c.Emit2(r.offset);

//want to store the value to that location

c.Emit1(LDLR4);

return true;

}

}

return false;

}

bool Parser::ReturnType(enum Type &returnType)

{

bool parsedCorrectly = false;

if (Stray(returnType))

{

c.Log("Type:Stray");

returnType = TypeStray;

parsedCorrectly = true;

}

else if (Type(returnType))

{

parsedCorrectly = true;

}

return parsedCorrectly;

}

//god help us

bool Parser::Doghouse()

{

size\_t s;

if(Match(DefaultToken))

{

return true;

}

else if(BlockStatement(s))

{

return true;

}

else if(Match(BreakToken))

{

return true;

}

return false;

}

bool Parser::BedStatements(size\_t &lines)

{

size\_t s, sum = 0;

while (!Match(BreakToken))

{

if (Statement(s))

{

sum += s;

}

else

return false;

}

lines = sum;

return true;

}

bool Parser::BedLabel()

{

if (Match(CaseToken))

{

if (Match(IntToken))

{

if (Match(BedEndToken))

{

return true;

}

}

}

return false;

}

bool Parser::Bed(enum Type t)

{

size\_t s;

if(BedLabel())

{

if(BedStatements(s))

{

if(Match(BreakToken))

{

//TODO jump to end of switch statement

c.Emit1(JMP);

c.Emit2(s);

//TODO build up address index array for switch statement

return true;

}

}

}

return false;

}

bool Parser::Beds(enum Type t)

{

if(Bed(t))

{

return true;

}

return false;

}

bool Parser::BlockBeds(enum Type t)

{

if(Doghouse())

{

return true;

}

else if(Beds(t))

{

return true;

}

return false;

}

bool Parser::NapTimeStatement()

{

enum Type t = TypeStray;

if(Match(SwitchToken))

{

if(Expr(t))

{

//TODO save value on stack

if(Match(LBlockToken))

{

if(BlockBeds(t))

{

//todo: everything

return true;

}

}

}

}

return false;

}

//done

bool Parser::Statement(size\_t &lines)

{

enum Type t = TypeStray;

if(FeedStatement())

{

// TODO feed statement

}

else if(SpeakStatement())

{

// TODO speak statement

}

else if(Expr(TypeStray))

{

return true;

}

else if(Loop())

{

return true;

}

else if(SniffStatement())

{

return true;

}

else if(NapTimeStatement())

{

return true;

}

else if (ReturnType(t))

{

//function or variable

size\_t dontcare;

this->Declaration(t, dontcare);

}

if (Match(EoLToken))

{

//hey!, end of a line, how cute

return true;

}

return false;

}

bool Parser::BlockStatement(size\_t &lines){

size\_t tmp, sum = 0;

if(Match(LBlockToken))

{

c.Log(" Block (");

while (!Match(RBlockToken))

{

Match(EoLToken);

sum += Statement(tmp);

Match(EoLToken);

}

c.Log(")");

return true;

}

else if(Statement(lines))

{

return true;

}

return false;

}

//done

bool Parser:: Loop()

{

if (RollOverLoop())

{

return true;

}

else if (ChaseLoop())

{

return true;

}

else if(ScratchLoop())

{

return true;

}

return false;

}

bool Parser::Stray(enum Type& t)

{

if (Match(VoidToken))

{

t = TypeStray;

return true;

}

return false;

}

bool Parser::Predicate()

{

enum Type evaluated = TypeDalmatian;

Expr(evaluated);

return evaluated == TypeDalmatian;

}

bool Parser::Parameters(vector<Parameter>& params)

{

enum Type t;

if (Match(LParenToken))

{

do {

if (Type(t))

{

if (currentToken.token == NameToken)

{

//we have a parameter

Parameter p;

p.type = t;

p.name = currentToken.value;

p.isArray = false;

p.numOfElements = 1;

params.push\_back(p);

GetNextToken();

}

}

} while (Match(SeparatorToken));

if (!Match(RParenToken)) {

//error

cout << "Expected closing bone, instead got " << currentToken.value;

return false;

}

return true;

}

return false;

}

bool Parser::FunctionDeclaration(enum Type returnType, size\_t& pcline)

{

size\_t size;

//add this so we can add parameters to it

vector<Parameter> params;

if (Parameters(params))

{

c.Log(" Function ( Params: ");

Match(RParenToken);

if (currentToken.token == NameToken)

{

//set acces link

//Record r =

//function being declared

table.AddFunction(currentToken.value, returnType);

//make access link

c.Emit1(PUSL);

c.Emit1(STLR2);

c.Emit2(table.MostCurrentStackSize());

//push L again

c.Emit1(PUSL);

bool isMain = false;

//check to see if it is the main function

if (currentToken.value == "walk") {

//this is the main function

pcline = c.CurrentAddr();

isMain = true;

}

//add all the parameters

for (size\_t i = 0; i < params.size(); ++i)

{

Parameter p = params[i];

c.Log(p.name + ":" + to\_string(p.type) + ",");

table.AddParameter(p.name, p.type, p.isArray, p.numOfElements);

}

c.Log(" ) Name: " + currentToken.value + " Body: (");

GetNextToken();

if (BlockStatement(size))

return true;

c.Log (" ) ) ");

if (isMain)

{

//add halt instruction to gtfo

c.Emit1(HALT);

}

}

}

return false;

}

bool Parser::Declaration(enum Type typeDeclared, size\_t& pcline)

{

c.Log(" Defined a " + to\_string(typeDeclared) + " ");

if (FunctionDeclaration(typeDeclared, pcline))

{

return true;

}

else if (currentToken.token == NameToken)

{

std::string name = currentToken.value;

GetNextToken();

VariableDeclaration(typeDeclared, name);

return true;

}

return false;

}

bool Parser::VariableEnding(enum Type t, std::string name)

{

//it was declared, we need enter the value

table.AddVariable(name, t);

if (Match(EoLToken))

{

c.Log(" End;");

return true;

}

else if (RHS(t, name))

{

return true;

}

return false;

}

bool Parser::RHSContinued(enum Type t)

{

if (Match(EoLToken))

{

c.Log("End of Line.\r\n");

//ending previous declaration

return true;

}

else if (Match(SeparatorToken))

{

c.Log(", ");

if (currentToken.token == NameToken)

{

string name = currentToken.value;

GetNextToken();

//read in next

VariableDeclaration(t, name);

return true;

}

}

return false;

}

//lil' helper

bool Parser::Match( enum Token t)

{

if (currentToken.token == t)

{

GetNextToken();

return true;

}

return false;

}

bool Parser::Literal(enum Type &t)

{

if(currentToken.token == IntToken)

{

int val = to\_int(currentToken.value);

GetNextToken();

c.Log(" (Int: " + to\_string(val) + ")");

//EMIT

c.Emit1(LDM4);

c.Emit4(val);

return true;

}

else if(currentToken.token == CharToken)

{

char tmp;

stringstream ss;

ss << currentToken.value;

ss >> tmp;

GetNextToken();

c.Log(" (Char: " + currentToken.value + ")");

//EMIT

c.Emit1(LDM1);

c.Emit1(tmp);

return true;

}

else if(currentToken.token == RealToken)

{

float tmp;

stringstream ss;

ss << currentToken.value;

ss >> tmp;

GetNextToken();

string toPrint = to\_string(tmp);

c.Log(" (Float: " + toPrint + ")");

//EMIT

c.Emit1(LDM4);

//convert this guy to an int, eh

float f = tmp;

int \*i;

i = (int \*) &f;

c.Emit4(\*i);

return true;

}

else if(currentToken.token == BoolToken)

{

char tmp;

if (currentToken.value == "XX")

tmp = 0;

else

tmp = 1;

GetNextToken();

string s = (tmp ? "OO" : "XX");

c.Log(" (Char: " + s + ")");

//EMIT

c.Emit1(LDM1);

c.Emit1(tmp);

return true;

}

return false;

}

bool Parser::Type(enum Type &t)

{

//if(Match(StringDecToken))

//{

// return true;

//}

if(Match(IntDecToken))

{

t = TypePoodle;

return true;

}

else if(Match(BoolDecToken))

{

t = TypeDalmatian;

return true;

}

else if(Match(RealDecToken))

{

t = TypeCorgi;

return true;

}

else if(Match(CharDecToken))

{

t = TypeYorkie;

return true;

}

return false;

}

bool Parser::Expr0(enum Type t)

{

if (Match(FetchToken))

{

c.Log ("Function (");

if (FunctionCall())

{

}

c.Log(")");

return true;

}

else if (Match(LParenToken))

{

c.Log(" Expr ( ");

if (Expr(t))

{

if (Match(RParenToken))

{

c.Log (" )");

return true;

}

else

cout << "ERROR" << endl;

return false;

}

}

else if (currentToken.token == NameToken)

{

string name = currentToken.value;

Record r = table.FindRecord(currentToken.value);

GetNextToken();

if (Match(ArrayDefToken))

{

c.Log(" Array[");

//go to where this is delcared

c.Emit1(PUSL);

//dereference to where

for (size\_t i = table.CurrentScope(); i > r.scope; ++i)

c.Emit1(DER);

if (Expr(TypePoodle))

{

if (Match(ArrayDefToken))

{

c.Log("]");

size\_t typesize = TypeSize(r.type);

if (typesize == 4)

{

//find scope this is declared in

//Find offset of value

c.Emit2(LDM4);

c.Emit4(4);

c.Emit1(MUL);

}

}

}

}

else

{

c.Log (" NameToken: " + name + " ");

c.Emit1(LDLR4);

c.Emit2(r.offset);

}

return true;

}

else if (Literal(t))

{

return true;

}

return false;

}

bool Parser::Expr1(enum Type t)

{

if (Match(NegateToken))

{

c.Log("Negate (");

if (Expr0(t))

{

}

c.Log(" )");

//emit and code

c.Emit1(NEG);

return true;

}

else if (Match(IncrToken))

{

c.Log("Increment (");

if (Expr0(t))

{

}

c.Log(" )");

c.Emit1(INC);

}

else

return Expr0(t);

return false;

}

bool Parser::Expr2(enum Type t)

{

if (Expr1(t))

{

if (Match(MultToken))

{

c.Log (" <3 ");

if (Expr2(t))

{

}

//emit and code

c.Emit1(MUL);

}

else if (Match(DivToken))

{

c.Log (" VV ");

if (Expr2(t))

{

}

//emit and code

c.Emit1(DIV);

}

else if (Match(ModToken))

{

c.Log (" {} ");

if (Expr2(t))

{

}

//emit and code

c.Emit1(MOD);

}

return true;

}

return false;

}

bool Parser::Expr3(enum Type t)

{

if (Expr2(t))

{

if (Match(AddToken))

{

c.Log (" + ");

if (Expr4(t))

{

}

//emit and code

c.Emit1(ADD);

}

else if (Match(SubToken))

{

c.Log (" - ");

if (Expr4(t))

{

}

//emit and code

c.Emit1(SUB);

}

return true;

}

return false;

}

bool Parser::Expr4(enum Type t)

{

if (Expr3(t))

{

if (Match(EqToken))

{

c.Log(" == ");

if (Expr4(t))

{

}

//emit and code

c.Emit1(SUB);

c.Emit1(TZE);

}

return true;

}

return false;

}

bool Parser::Expr6(enum Type t)

{

if (Expr5(t))

{

if (Match(AssignToken))

{

c.Log (" Assign ");

if (Expr5(t))

{

}

//TODO find record location

//TODO store value in record

}

return true;

}

return false;

}

bool Parser::Expr5(enum Type t)

{

if (Expr4(t))

{

if (Match(AndToken))

{

c.Log (" AND ");

if (Expr5(t))

{

}

//emit and code

c.Emit1(AND);

}

return true;

}

return false;

}

bool Parser::FunctionCall()

{

if (Match(FetchToken))

{

}

return false;

}

bool Parser::Expr(enum Type t)

{

if (Expr6(t)) {

//do some shit

if (Match(OrToken))

{

c.Log (" OR ");

if (Expr(t))

{

}

}

return true;

}

return false;

}

bool Parser::RHS(enum Type t, std::string idName)

{

if (Match(AssignToken))

{

c.Log("Assign (");

this->Expr(t);

//then we need to assign the value to the variable

Record r = table.FindRecord(idName);

//want to push the memory location onto the stack

c.Emit1(LDM2);

c.Emit2(r.offset);

//want to store the value to that location

c.Emit1(STLR4);

c.Log(") to (" + r.name + " )");

return this->RHSContinued(t);

}

return false;

}

bool Parser::VariableDeclaration(enum Type t, std::string idName)

{

c.Log( "\""+ idName + "\" ");

if (Match(ArrayDefToken))

{

c.Log(" [");

int value;

if (Poodle(value))

{

if (Match(ArrayDefToken))

{

//was able to parse the entire array definition

table.AddArray(idName, t, value);

c.Log(to\_string(value) + "]");

}

}

return true;

}

else if (VariableEnding(t, idName))

return true;

else {

//ERROR

}

return false;

}

bool Parser::Declarations(size\_t &pcline)

{

enum Type typeDeclared;

while (s.HasToken())

{

if (Match( EoLToken))

{

//parse the other declarations, no code gen

c.Log("EoL");

}

else if (ReturnType(typeDeclared))

{

//function or variable

this->Declaration(typeDeclared, pcline);

}

}

return false;

}

//done on speaks

bool Parser::SpeakStatement ()

{

if (SpeakPoodle())

{

return true;

}

else if (SpeakCorgi())

{

return true;

}

else if(SpeakYorkie())

{

return true;

}

else if (SpeakDalmatian())

{

return true;

}

return false;

}

bool Parser:: SpeakPoodle()

{

if(Match(PrintIntToken))

{

//output tree to screen, if wanted

c.Log("SpeakPoodle (");

//try to evaluate the following expression

if(Expr(TypePoodle))

{

c.Emit1(OUTI);

//then output a new line

c.Emit1(LDM1);

c.Emit1('\n');

c.Emit1(OUTC);

c.Log(")");

return true;

}

}

return false;

}

bool Parser:: SpeakCorgi()

{

if(Match(PrintRealToken))

{

c.Log("SpeakCorgi (");

if(Expr(TypeCorgi))

{

c.Emit4(OUTR);

c.Log(")");

}

}

return false;

}

bool Parser:: SpeakYorkie()

{

if(Match(PrintCharToken))

{

c.Log("SpeakYorkie(");

if(Expr(TypeYorkie))

{

c.Emit1(OUTC);

c.Log(")");

}

}

return false;

}

bool Parser:: SpeakDalmatian()

{

if(Match(PrintBoolToken))

{

c.Log("SpeakDalmatian(");

if(Expr(TypeDalmatian))

{

c.Emit1(TNZ);

c.Emit1(BYES);

size\_t falsebranch = c.Emit2(0);

//branch to the false case, falls through to the true

c.Emit1(LDM1);

c.Emit1('X');

c.Emit1(LDM1);

c.Emit1('X');

c.Emit1(OUTC);

c.Emit1(OUTC);

c.Emit1(JMPD);

size\_t wheretheEndGoes = c.Emit2(0);

size\_t startOfFalse = c.Emit1(LDM1);

c.Emit1('O');

c.Emit1(LDM1);

c.Emit1('O');

c.Emit1(OUTC);

c.Emit1(OUTC);

size\_t endOfFalsebranch = c.CurrentAddr();

//backpatch like a boss

c.BackPatch2(falsebranch, startOfFalse-falsebranch);

c.BackPatch2(wheretheEndGoes, endOfFalsebranch);

}

else

{

return false;

}

}

return false;

}

//TODO: care

bool Parser:: RollOverLoop()

{

int dontcare;

size\_t blockSize;

if(Match(ForToken))

{

c.Log ("ForToken (");

enum Type t;

if (Type(t))

{

if(currentToken.token == NameToken)

{

c.Log(" Var:" + to\_string(t) + " \"" + currentToken.value + "\" ");

string name = currentToken.value;

//add record

if (!table.AddVariable(currentToken.value, t))

//record wasn't able to be added

//ERROR

return false;

Record r = table.FindRecord(currentToken.value);

GetNextToken();

if(Match(FromToken))

{

c.Log(" From ");

if (Poodle(dontcare)) //TODO Evaluate the poodles

{

//store value on stack, to record location

c.Emit1(STLR4);

c.Emit2(r.offset);

if(Match(ToToken))

{

c.Log (" To ");

//put it back on the stack

size\_t predicateStart = c.CurrentAddr();

c.Emit1(LDLR4);

c.Emit2(r.offset);

if (Poodle(dontcare))

{

c.Emit1(SUB);

c.Emit1(TNG);

c.Emit1(BNO);

size\_t toBeBackPatched = c.Emit2(0);

//determine if we should the loop

if(BlockStatement(blockSize))

{

//increment step

c.Emit1(LDLR4);

c.Emit2(r.offset);

c.Emit1(INC);

c.Emit1(STLR4);

c.Emit2(r.offset);

//compare to see if still valid, just go back up to predicate

c.Emit1(JMP);

c.Emit2(toBeBackPatched);

size\_t endOfBlock = c.CurrentAddr();

c.BackPatch4(toBeBackPatched, endOfBlock - toBeBackPatched);

}

}

}

}

}

}

}

}

return false;

}

//done

bool Parser:: ChaseLoop()

{

size\_t blockSize;

if (Match(WhileToken))

{

c.Log("WhileToken (");

size\_t predicateAddress = c.CurrentAddr();

if (Predicate())

{

c.Emit1(BNO);

size\_t toBeBackPatched = c.Emit2(0);

if (BlockStatement(blockSize))

{

//TODO

c.Emit1(JMP);

c.Emit2(predicateAddress);

size\_t endOfBlock = c.CurrentAddr();

c.BackPatch2(toBeBackPatched, endOfBlock-toBeBackPatched);

}

else

return false;

}

c.Log(")");

}

return false;

}

//done maybe

bool Parser::FeedStatement()

{

if(FeedPoodle())

{

return true;

}

else if(FeedCorgi())

{

return true;

}

else if(FeedYorkie())

{

return true;

}

else if(FeedDalmatian())

{

return true;

}

return false;

}

bool Parser::FeedPoodle()

{

if(Match(InputIntToken))

{

c.Log("FeedPoodle (");

enum Type typeOfVar = TypeStray;

if(Expr0(typeOfVar))

{

if (typeOfVar != TypePoodle)

{

//ERROR, try to read in into to non-interger

return false;

}

else

{

//variable was of type poodle, read in number

c.Emit2(INPUTI);

c.Log(")");

}

//comment odo

}

}

return false;

}

bool Parser::FeedCorgi()

{

if(Match(InputRealToken))

{

c.Log("FeedCorgi (");

enum Type typeOfVar = TypeStray;

if(Expr0(typeOfVar))

{

if(typeOfVar != TypeCorgi)

{

//Error

return false;

}

else

{

//variable was type Corgi, read in number

c.Emit2(INPUTR);

c.Log(")");

}

}

}

return false;

}

bool Parser::FeedYorkie()

{

if(Match(InputIntToken))

{

c.Log("FeedYorkie (");

enum Type typeOfVar = TypeStray;

if(Expr0(typeOfVar))

{

if(typeOfVar != TypeYorkie)

{

//Error

return false;

}

else

{

//variable was type Yotkie, read in number

c.Emit2(INPUTC);

c.Log(")");

}

}

}

return false;

}

bool Parser::FeedDalmatian()

{

if(Match(InputIntToken))

{

c.Log("FeedDalmatian (");

enum Type typeOfVar = TypeStray;

if(Expr0(typeOfVar))

{

if(typeOfVar != TypeDalmatian)

{

//Error

return false;

}

if(c.Emit2(INPUTC) == 0)

{

c.Emit1(STLR1);

c.Log(")");

return true;

}

else if(c.Emit2(INPUTC) == 1)

{

c.Emit1(STLR1);

c.Log(")");

return true;

}

//comment todo

}

}

return false;

}

bool Parser::ScratchLoop()

{

size\_t s;

if(Match(RepeatToken))

{

size\_t predicateAddress = c.CurrentAddr();

c.Log("ScratchLoop (");

if(BlockStatement(s))

{

c.Emit1(JMP);

c.Emit2(predicateAddress);

size\_t endOfBlock = c.CurrentAddr();

if(Match(UntilToken))

{

c.Log("Until (");

if(Predicate())

{

//comment todo

c.Emit1(BNO);

size\_t toBeBackPatched = c.Emit2(0);

c.BackPatch2(toBeBackPatched, endOfBlock-toBeBackPatched);

}

}

}

}

return false;

}

bool Parser::SniffStatement()

{

size\_t s;

if(Match(IfToken))

{

c.Log("If (");

size\_t predicateAddress = c.CurrentAddr();

if(Predicate())

{

c.Emit1(BNO);

size\_t toBeBackPatched = c.Emit2(0); //placeholder

//we evaluated the expresion, only continue if it was true

if(BlockStatement(s))

{

if (Match(ElseToken))

{

c.Log("Else (");

c.Emit1(JMP);

size\_t toBeEndOfElse = c.Emit2(0); //placeholder

size\_t beginningOfElse = c.CurrentAddr();

if(BlockStatement(s))

{

//else block was parsed

c.BackPatch2(toBeBackPatched, beginningOfElse-toBeBackPatched); // set else brach to go here

c.BackPatch2(toBeEndOfElse , beginningOfElse-toBeEndOfElse); // set if branch to leave body altogether

}

}

else

{

//was not an else statement

//backpatch to this location]

size\_t endOfBlock = c.CurrentAddr();

c.BackPatch2(toBeBackPatched, endOfBlock-toBeBackPatched);

}

}

else

return false;

}

}

return false;

}

bool Parser::Comparator(){

if (currentToken.token == LessToken)

{

return true;

}

else if (currentToken.token == GreaterToken)

{

return true;

}

else if (currentToken.token == EqToken)

{

return true;

}

else if (currentToken.token == NotEqualToken)

{

return true;

}

else if (currentToken.token == GreaterOrEqualToken)

{

return true;

}

else if (currentToken.token == LessOrEqualToken)

{

return true;

}

return false;

}

void Parser::Run()

{

//start of program

c.Emit1(LDM2);

size\_t PCLocation = c.Emit2(0);

c.Emit1(POPP);

size\_t PClineNum =0;

//start reading tokens

GetNextToken();

Declarations(PClineNum);

//backpatch the start of the program

c.BackPatch2(PCLocation, PClineNum);

//EndProgram

c.Emit1(HALT);

c.Emit1(HALT);

c.Emit1(HALT);

c.Emit1(HALT);

}

void Parser::Flush()

{

c.flush();

}

//Paser Header File

#ifndef PARSER\_H

#define PARSER\_H

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

#include "scanner.h"

#include "tokens.h"

#include "symTable.h"

#include "doggieTypes.h"

#include "Interpreter.h"

#include "codegen.h"

#include <iostream>

#include <string>

#include <vector>

class Parser

{

private:

Scanner s;

SymTable table;

cg c;

TokenInfo currentToken;

TokenInfo GetNextToken();

//this will compare the current token to t, if they are the same

//it will grab the next token and return true. Otherwise

//returns false.

bool Match(enum Token t);

//Below are methods that try to consume tokens to match their definition

bool BedLabel();

bool Declarations(size\_t&);

bool Parameters(std::vector<Parameter>&);

bool ArrayLHS(enum Type, std::string);

bool Statements();

bool Loop();

bool Doghouse();

bool NapTimeStatement();

bool FeedPoodle();

bool FeedCorgi();

bool FeedYorkie();

bool FeedDalmatian();

bool ScratchLoop();

bool FeedStatement();

bool SniffStatement();

/\*bool SneezeStatement();\*/

bool SpeakStatement();

bool SpeakPoodle();

bool SpeakCorgi();

bool SpeakYorkie();

/\*bool SpeakDachshund();\*/

bool SpeakDalmatian();

bool RollOverLoop();

bool ChaseLoop();

bool Comparator();

bool FunctionCall();

//these methods will try to fulfill some type requirement or

//the operation needs additional information to implement.

bool FunctionDeclaration(enum Type, size\_t&);

bool Declaration(enum Type, size\_t&);

bool ArrayDeclaration(enum Type);

bool VariableEnding(enum Type, std::string n);

bool RHS(enum Type, std::string);

bool RHSContinued(enum Type);

//pass TypeStray if the type is irrelevant

bool Expr6(enum Type);

bool Expr5(enum Type);

bool Expr4(enum Type);

bool Expr3(enum Type);

bool Expr2(enum Type);

bool Expr1(enum Type);

bool Expr0(enum Type);

bool Expr(enum Type);

bool Beds(enum Type);

bool Bed(enum Type);

bool BlockBeds(enum Type);

bool Predicate();

bool VariableDeclaration(enum Type, std::string);

//these methods return the type which they were able to determine.

bool Literal(enum Type&);

bool ReturnType(enum Type&);

bool Stray(enum Type&);

bool Type(enum Type&);

bool Poodle();

bool Poodle(int&); //checks for literal

bool Dalmatian(enum Type&);

bool Corgi(enum Type&);

bool Yorkie(enum Type&);

bool TypeDeclaration(enum Type&);

bool Statement(size\_t&); //reads just a single line

bool BlockStatement(size\_t&);//reads-in a single line, or block

bool BedStatements(size\_t&); //reads in lines until a break

public:

Parser(std::iostream \*stream, std::ostream& out) : s(stream), c(out) { };

void Run();

void Flush();

};

#endif

//ParserTests

//Unit Test File

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

#include <iostream>

#include <sstream>

#include "QUnit.hpp"

int main()

{

QUnit::UnitTest qunit(std::cerr, QUnit::normal);

return 0;

}

//QUnit

//QUnit.hpp - a simple unit test framework for C++

//

// Typical usage:

//

// #include "QUnit.hpp"

//

// int main() {

// QUnit::UnitTest qunit(std::cerr, QUnit::verbose);

//

// QUNIT\_IS\_TRUE(true);

// QUNIT\_IS\_FALSE(4!=4);

// QUNIT\_IS\_EQUAL(42, 42.0);

// QUNIT\_IS\_NOT\_EQUAL(42,"43");

//

// return qunit.errors();

// }

//

#ifndef \_QUNIT\_HPP\_

#define \_QUNIT\_HPP\_

#include <sstream>

#include <string>

#define QUNIT\_IS\_EQUAL(expr1,expr2) QUNIT\_COMPARE(true,true,expr1,expr2)

#define QUNIT\_IS\_NOT\_EQUAL(expr1,expr2) QUNIT\_COMPARE(true,false,expr1,expr2)

#define QUNIT\_IS\_TRUE(expr) QUNIT\_COMPARE(false,true,expr,true)

#define QUNIT\_IS\_FALSE(expr) QUNIT\_COMPARE(false,true,expr,false)

#define QUNIT\_COMPARE(compare,result,expr1,expr2) { \

std::stringstream s1, s2; \

s1 << std::boolalpha << (expr1); \

s2 << std::boolalpha << (expr2); \

qunit.evaluate( \

compare, result, s1.str(), s2.str(), #expr1, #expr2, \

\_\_FILE\_\_, \_\_LINE\_\_, \_\_FUNCTION\_\_ ); \

}; \

namespace QUnit {

enum { silent, quiet, normal, verbose, noisy };

class UnitTest {

public:

UnitTest(std::ostream & out, int verboseLevel);

~UnitTest();

void verboseLevel(int level);

int verboseLevel();

void printStatus();

int errors() const;

void evaluate(bool, bool,

std::string, std::string, std::string, std::string,

const char \*, int, const char \*);

private:

int verboseLevel\_;

int errors\_;

int tests\_;

std::ostream & out\_;

};

inline UnitTest::UnitTest(std::ostream & out, int verboseLevel)

: verboseLevel\_(verboseLevel) , errors\_(0) , tests\_(0) , out\_(out) {

}

inline UnitTest::~UnitTest() {

if ( verboseLevel\_ > quiet )

printStatus();

}

inline void UnitTest::verboseLevel(int level) {

verboseLevel\_ = level;

}

inline int UnitTest::verboseLevel() {

return verboseLevel\_;

}

inline void UnitTest::printStatus() {

out\_ << "Testing " << ( errors\_ ? "FAILED" : "OK" ) << " ("

<< tests\_ << " tests, " << ( tests\_ - errors\_ ) << " ok, "

<< errors\_ << " failed)" << std::endl;

}

inline int UnitTest::errors() const {

return errors\_;

}

inline void UnitTest::evaluate(

bool compare, bool result,

std::string val1, std::string val2,

std::string str1, std::string str2,

const char \* file, int line, const char \* func) {

bool ok = result ? (val1 == val2) : (val1 != val2);

tests\_ += 1;

errors\_ += ok ? 0 : 1;

if( (ok && !(verboseLevel\_ > normal)) || verboseLevel\_ == silent )

return;

out\_ << file << ( ok ? ";" : ":" ) << line << ": ";

out\_ << ( ok ? "OK/" : "FAILED/" ) << func << "(): ";

if( compare ) {

const std::string cmp = ( result ? "==" : "!=" );

out\_ << "compare {" << str1 << "} " << cmp << " {" << str2 << "} "

<< "got {\"" << val1 << "\"} " << cmp << " {\"" << val2 << "\"}";

} else {

out\_ << "evaluate {" << str1 << "} == " << val1;

}

out\_ << std::endl;

}

}

#endif // \_QUNIT\_HPP\_

//scanner

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

#include "scanner.h"

#include <iostream>

#include <sstream>

#include <cctype>

#include <string>

template <typename T>

std::string patch::to\_string(T var)

{

std::stringstream ss;

ss << var;

return ss.str();

}

char Scanner::GetNext()

{

\*input >> next;

currentToken.value.push\_back(next);

next = input->peek();

return next;

}

char Scanner::GetFirst()

{

ConsumeWS();

next = input->peek();

return next;

}

void Scanner::ReadEscapedCharacter()

{

if (next == '\*') {

//skip escape literal, read in the next character

\*input >> next;

next = input->peek();

//read in value

char escaped = '\0';

if (next == '\*' || next == '$' || next == '^' || next == '\n')

escaped = next;

else if (next == '@')

{

//read in hex value

\*input >> next; //read in @, as it's not the value we need;

\*input >> std::hex >> escaped;

}

else if (next >= '0' && next < '9')

{

//read octal

\*input >> std::oct >> escaped;

}

else if (next == 'r')

escaped = '\r';

else if (next == 'n')

escaped = '\n';

else if (next == '?')

escaped = '\?';

else if (next == 't')

escaped = '\t';

else if (next == 'v')

escaped = '\v';

else if (next == 'f')

escaped = '\f';

else if (next == 'b')

escaped = '\b';

else if (next == 'a')

escaped = '\a';

else {

std::cout << "Tried to read in improper escaped character \'\*" << next << "\' on line " << lineNumber << std::endl;

throw 0;

}

//if it's a new line, then continue reading the multiline string

if (escaped != '\n')

currentToken.value.push\_back(escaped);

\*input >> next; //old value still on buffer, remove

next = input->peek(); //set up for next match

}

}

void Scanner::ParseString()

{

if (next == '$')

{

//read in the $ from input, but discard it because it's not lexically important

GetNext();

currentToken.value = "";

while (next != '$' && next != '\n')

{

if (next == '\*')

this->ReadEscapedCharacter();

else

GetNext();

}

//loop was broken by a newline,

if (next == '\n') {

std::cout << "String wasn't closed, use \* at the end of the line for multiline strings. Line: "<< lineNumber << std::endl;

throw 0;

}

currentToken.token = StringLitToken;

//consume lingering $

\*input >> next;

next = input->peek();

}

}

void Scanner::ParseNumber()

{

//consume all of the digits for an int

while (isdigit(GetNext()))

;

//check to see if it is a real

if (next != '.')

{

//not a real

currentToken.token = IntToken;

}

else {

//consume the rest of the numbers

while (isdigit(GetNext()))

;

currentToken.token = RealToken;

}

}

void Scanner::ConsumeWS()

{

while (isspace(input->peek()))

{

if (next == '\n' || next == '\r')

++lineNumber;

SkipNext();

}

}

inline void Scanner::SkipNext()

{

\*input >> std::ws;

next = input->peek();

}

bool Scanner::HasToken()

{

ConsumeWS();

return !input->eof() && !hasErrored;

}

TokenInfo Scanner::NextToken()

{

currentToken.token = ErrorToken; //This is the default case

currentToken.value = ""; //Clear previous lexeme

GetFirst();

switch (next)

{

case '#':

currentToken.token = AddToken;

GetNext();

break;

case '\_':

GetNext();

if(next == '/')

{

currentToken.token = SubToken;

GetNext();

}

else

currentToken.token = ErrorToken;

break;

case '<':

GetNext();

if(next == '3')

{

currentToken.token = MultToken;

GetNext();

}

else if(next == '}')

{

GetNext();

if(next == '<')

{

GetNext();

if(next == '>')

{

currentToken.token = LessOrEqualToken;

GetNext();

}

}

else

{

currentToken.token = LessToken;

GetNext();

}

}

else if(next == '>')

{

currentToken.token = EqToken;

GetNext();

}

else

currentToken.token = ErrorToken;

break;

case 'V':

GetNext();

if(next == 'V')

{

currentToken.token = DivToken;

GetNext();

}

else

currentToken.token = ErrorToken;

break;

case '{':

GetNext();

if(next == '}')

{

currentToken.token = ModToken;

GetNext();

}

else if(next == '>')

{

GetNext();

if(next == '<')

{

GetNext();

if(next == '>')

{

currentToken.token = GreaterOrEqualToken;

GetNext();

}

}

else

{

currentToken.token = GreaterToken;

GetNext();

}

}

else

currentToken.token = ErrorToken;

break;

case '/':

GetNext();

if(next == 'T')

{

currentToken.token = IncrToken;

GetNext();

}

else if(next == '|')

{

currentToken.token = AssignToken;

GetNext();

}

else if(next == '/')

{

currentToken.token = OrToken;

GetNext();

}

else

currentToken.token = ErrorToken;

break;

case '\\':

GetNext();

if(next == 'T')

{

currentToken.token = DecrToken;

GetNext();

}

else

currentToken.token = ErrorToken;

break;

case ':':

GetNext();

if(next == '3')

{

GetNext();

if(next == '<')

{

GetNext();

if(next == '>')

{

currentToken.token = NotEqualToken;

GetNext();

}

}

else

{

currentToken.token = NegateToken;

GetNext();

}

}

else if(next == ':')

{

currentToken.token = SeparatorToken;

GetNext();

}

else

currentToken.token = ErrorToken;

break;

case 'O':

GetNext();

if(next == 'O')

{

currentToken.token = TrueToken;

GetNext();

}

else

currentToken.token = ErrorToken;

break;

case 'X':

GetNext();

if(next == 'X')

{

currentToken.token = FalseToken;

GetNext();

}

else

currentToken.token = ErrorToken;

break;

case '@':

GetNext();

if (next == '=') {

currentToken.token = LParenToken;

GetNext();

}

else

currentToken.token = HexToken;

break;

case '^':

currentToken.token = CharLitToken;

GetNext();

break;

case '$':

this->ParseString();

break;

case '+':

GetNext();

if(next == '&')

{

currentToken.token = PositiveToken;

GetNext();

}

else

currentToken.token = ErrorToken;

break;

case '?':

currentToken.token = BedEndToken;

GetNext();

break;

case '\*':

currentToken.token = StringEscToken;

GetNext();

break;

case '8':

case '0':

case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '9':

this->ParseNumber();

break;

case '=':

GetNext();

if(next == '@')

{

currentToken.token = RParenToken;

GetNext();

}

else

currentToken.token = ErrorToken;

break;

case '|':

GetNext();

if(next == 'P')

{

currentToken.token = AddressOfToken;

GetNext();

}

else

{

currentToken.token = ArrayDefToken;

GetNext();

}

break;

case '.':

GetNext();

if(next == '.')

{

currentToken.token = PointerDefToken;

GetNext();

}

//could be a real number, ex. .67

else if (isdigit(next))

{

//push back on the . and number, then parse has number

std::stringstream ss;

ss << '.';

ss << next;

ss << \*input;

input->clear();

\*input << ss;

currentToken.value = "";

this->ParseNumber();

}

else

currentToken.token = ErrorToken;

break;

case '-':

GetNext();

if(next == '-')

{

currentToken.token = DerefToken;

GetNext();

}

else if(next == '&')

{

currentToken.token = NegativeToken;

GetNext();

}

else

currentToken.token = ErrorToken;

break;

case '!':

currentToken.token = EoLToken;

GetNext();

break;

case 'G':

GetNext();

if(next == 'R')

{

GetNext();

if(next == 'R')

{

currentToken.token = LMultiLineCommentToken;

GetNext();

break;

}

}

currentToken.token = ErrorToken;

break;

case 'B':

GetNext();

if(next == 'A')

{

GetNext();

if(next == 'R')

{

GetNext();

if(next == 'K')

{

currentToken.token = RMultiLineCommentToken;

GetNext();

break;

}

}

}

else if(next == 'e')

{

GetNext();

if(next == 'd')

{

currentToken.token = CaseToken;

GetNext();

break;

}

}

currentToken.token = ErrorToken;

break;

case 'F':

GetNext();

if (next == 'e')

{

GetNext();

if (next == 'e')

{

GetNext();

if (next == 'd')

{

GetNext();

if (next == 'C')

{

GetNext();

if (next == 'o')

{

GetNext();

if (next == 'r')

{

GetNext();

if (next == 'g')

{

GetNext();

if (next == 'i')

{

currentToken.token = InputRealToken;

GetNext();

break;

}

}

}

}

}

else if (next == 'D')

{

GetNext();

if (next == 'a')

{

/\* GetNext();

if (next == 'c')

{

GetNext();

if (next == 'h')

{

GetNext();

if (next == 's')

{

GetNext();

if (next == 'h')

{

GetNext();

if (next == 'u')

{

GetNext();

if (next == 'n')

{

GetNext();

if (next == 'd')

{

currentToken.token = InputStringToken;

GetNext();

break;

}

}

}

}

}

}

}\*/

if (next == 'l')

{

GetNext();

if (next == 'm')

{

GetNext();

if (next == 'a')

{

GetNext();

if (next == 't')

{

GetNext();

if (next == 'i')

{

GetNext();

if (next == 'a')

{

GetNext();

if (next == 'n')

{

currentToken.token = InputBoolToken;

GetNext();

break;

}

}

}

}

}

}

}

}

}

else if (next == 'P')

{

GetNext();

if (next == 'o')

{

GetNext();

if (next == 'o')

{

GetNext();

if (next == 'd')

{

GetNext();

if (next == 'l')

{

GetNext();

if (next == 'e')

{

currentToken.token = InputIntToken;

GetNext();

break;

}

}

}

}

}

}

else if (next =='Y')

{

GetNext();

if (next =='o')

{

GetNext();

if (next =='r')

{

GetNext();

if (next =='k')

{

GetNext();

if (next =='i')

{

GetNext();

if (next =='e')

{

currentToken.token = InputCharToken;

GetNext();

break;

}

}

}

}

}

}

}

}

else if (next == 't') {

GetNext();

if (next == 'c') {

GetNext();

if (next == 'h') {

currentToken.token = FetchToken;

GetNext();

break;

}

}

}

}

else if (next == 'r')

{

GetNext();

if (next == 'o')

{

GetNext();

if (next == 'm')

{

currentToken.token = FromToken;

GetNext();

break;

}

}

}

currentToken.token = ErrorToken;

break;

case 'S':

GetNext();

if (next == 'c')

{

GetNext();

if (next == 'r')

{

GetNext();

if (next == 'a')

{

GetNext();

if (next == 't')

{

GetNext();

if (next == 'c')

{

GetNext();

if (next == 'h')

{

currentToken.token = RepeatToken;

GetNext();

break;

}

}

}

}

}

}

else if (next == 'n')

{

GetNext();

if (next == 'e')

{

GetNext();

if (next == 'e')

{

GetNext();

if (next == 'z')

{

GetNext();

if (next == 'e')

{

currentToken.token = ElseToken;

GetNext();

break;

}

}

}

}

else if (next == 'i')

{

GetNext();

if (next == 'f')

{

GetNext();

if (next == 'f')

{

currentToken.token = IfToken;

GetNext();

break;

}

}

}

}

else if (next == 'p')

{

GetNext();

if (next == 'e')

{

GetNext();

if (next == 'a')

{

GetNext();

if (next == 'k')

{

GetNext();

if (next == 'C')

{

GetNext();

if (next == 'o')

{

GetNext();

if (next == 'r')

{

GetNext();

if (next == 'g')

{

GetNext();

if (next == 'i')

{

currentToken.token = PrintRealToken;

GetNext();

break;

}

}

}

}

}

else if (next == 'D')

{

GetNext();

if (next == 'a')

{

/\*GetNext();

if (next == 'c')

{

GetNext();

if (next == 'h')

{

GetNext();

if (next == 's')

{

GetNext();

if (next == 'h')

{

GetNext();

if (next == 'u')

{

GetNext();

if (next == 'n')

{

GetNext();

if (next == 'd')

{

currentToken.token = PrintStringToken;

GetNext();

break;

}

}

}

}

}

}

}\*/

if (next == 'l')

{

GetNext();

if (next == 'm')

{

GetNext();

if (next == 'a')

{

GetNext();

if (next == 't')

{

GetNext();

if (next == 'i')

{

GetNext();

if (next == 'o')

{

GetNext();

if (next == 'n')

{

currentToken.token = PrintBoolToken;

GetNext();

break;

}

}

}

}

}

}

}

}

}

else if (next == 'P')

{

GetNext();

if (next == 'o')

{

GetNext();

if (next == 'o')

{

GetNext();

if (next == 'd')

{

GetNext();

if (next == 'l')

{

GetNext();

if (next == 'e')

{

currentToken.token = PrintIntToken;

GetNext();

break;

}

}

}

}

}

}

else if (next =='Y')

{

GetNext();

if (next =='o')

{

GetNext();

if (next =='r')

{

GetNext();

if (next =='k')

{

GetNext();

if (next =='i')

{

GetNext();

if (next =='e')

{

currentToken.token = PrintCharToken;

GetNext();

break;

}

}

}

}

}

}

}

}

}

}

else if (next == 'Q')

{

GetNext();

if (next == 'U')

{

GetNext();

if (next == 'I')

{

GetNext();

if (next == 'R')

{

GetNext();

if (next == 'R')

{

GetNext();

if (next == 'E')

{

GetNext();

if (next == 'L')

{

currentToken.token = LBlockToken;

GetNext();

break;

}

}

}

}

}

}

}

else if (next == 't')

{

GetNext();

if (next == 'r')

{

GetNext();

if (next == 'a')

{

GetNext();

if (next == 'y')

{

currentToken.token = VoidDecToken;

GetNext();

break;

}

}

}

}

currentToken.token = ErrorToken;

break;

case 'T':

GetNext();

if (next == 'o')

{

currentToken.token = ToToken;

GetNext();

break;

}

case 'N':

GetNext();

if(next == '\'')

{

currentToken.token = AndToken;

GetNext();

break;

}

else if (next == 'A')

{

GetNext();

if (next == 'P')

{

GetNext();

if (next == 'T')

{

GetNext();

if (next == 'I')

{

GetNext();

if (next == 'M')

{

GetNext();

if (next == 'E')

{

currentToken.token = SwitchToken;

GetNext();

break;

}

}

}

}

}

}

else if (next == 'e')

{

GetNext();

if (next == 'v')

{

GetNext();

if (next == 'e')

{

GetNext();

if (next == 'r')

{

GetNext();

if (next == 'm')

{

GetNext();

if (next == 'i')

{

GetNext();

if (next == 'n')

{

GetNext();

if (next == 'd')

{

GetNext();

if (next == '.')

{

GetNext();

if (next == '.')

{

GetNext();

if (next == '.')

{

currentToken.token = RBlockToken;

GetNext();

break;

}

}

}

}

}

}

}

}

}

}

}

else if ('O')

{

GetNext();

if (next == 'F')

{

GetNext();

if (next == 'L')

{

GetNext();

if (next == 'E')

{

GetNext();

if (next == 'A')

{

GetNext();

if (next == 'S')

{

currentToken.token = UntilToken;

GetNext();

break;

}

}

}

}

}

}

currentToken.token = ErrorToken;

break;

case 'D':

GetNext();

if(next=='a'){

GetNext();

/\*if(next=='c'){

GetNext();

if(next=='h'){

GetNext();

if(next=='s'){

GetNext();

if(next=='h'){

GetNext();

if(next=='u'){

GetNext();

if(next=='n'){

GetNext();

if(next=='d'){

currentToken.token=StringDecToken;

GetNext();

}

else

currentToken.token=ErrorToken;

break;

}

}

}

}

}

}\*/

if(next=='l'){

GetNext();

if(next=='m'){

GetNext();

if(next=='a'){

GetNext();

if(next=='t'){

GetNext();

if(next=='i'){

GetNext();

if(next=='a'){

GetNext();

if(next=='n'){

currentToken.token=BoolDecToken;

GetNext();

break;

}

else

currentToken.token=ErrorToken;

break;

}

}

}

}

}

}

}

else if (next == 'o'){

GetNext();

if(next == 'g'){

GetNext();

if(next == 'h'){

GetNext();

if(next == 'o'){

GetNext();

if(next == 'u'){

GetNext();

if(next == 's'){

GetNext();

if(next == 'e'){

currentToken.token=DefaultToken;

GetNext();

}

else

currentToken.token=ErrorToken;

break;

}

}

}

}

}

}

currentToken.token = ErrorToken;

break;

case 'P':

GetNext();

if(next=='A'){

GetNext();

if(next=='N'){

GetNext();

if(next=='T'){

currentToken.token=SingleLineCommentToken;

GetNext();

//read in the rest of the line

while(next!='\n')

GetNext();

}

else

currentToken.token=ErrorToken;

break;

}

}

else if(next=='o'){

GetNext();

if(next=='o'){

GetNext();

if(next=='d'){

GetNext();

if(next=='l'){

GetNext();

if(next=='e'){

currentToken.token=IntDecToken;

GetNext();

}

else

currentToken.token=ErrorToken;

break;

}

}

}

}

break;

case 'Y':

GetNext();

if(next=='o'){

GetNext();

if(next=='r'){

GetNext();

if(next=='k'){

GetNext();

if(next=='i'){

GetNext();

if(next=='e'){

currentToken.token=CharDecToken;

GetNext();

}

else

currentToken.token=ErrorToken;

break;

}

}

}

}

currentToken.token = ErrorToken;

break;

case 'C':

GetNext();

if(next=='h'){

GetNext();

if(next=='a'){

GetNext();

if(next=='s'){

GetNext();

if(next=='e'){

currentToken.token=WhileToken;

GetNext();

}

else

currentToken.token=ErrorToken;

break;

}

}

}

else if(next=='o'){

GetNext();

if(next=='r'){

GetNext();

if(next=='g'){

GetNext();

if(next=='i'){

currentToken.token=RealDecToken;

GetNext();

}

else

currentToken.token=ErrorToken;

break;

}

}

}

break;

case 'R':

GetNext();

if(next=='o'){

GetNext();

if(next=='l'){

GetNext();

if(next=='l'){

GetNext();

if(next=='o'){

GetNext();

if(next=='v'){

GetNext();

if(next=='e'){

GetNext();

if(next=='r'){

currentToken.token=ForToken;

GetNext();

}

else

currentToken.token=ErrorToken;

break;

}

}

}

}

}

}

break;

case 'W':

GetNext();

if(next == 'a'){

GetNext();

if(next == 'k'){

GetNext();

if(next == 'e'){

currentToken.token=BreakToken;

GetNext();

}

else

currentToken.token=ErrorToken;

break;

}

}

default:

//if it begins with a lower case, could be a name of a func or var

if (islower(next)) {

do {

GetNext();

} while ( (isalpha(next) && !isspace(next)) || isdigit(next));

currentToken.token = NameToken;

}

break;

}

if (currentToken.token == ErrorToken) {

currentToken.value = "Unkown symbol \"" + currentToken.value + "\" found on line " + patch::to\_string(lineNumber);

hasErrored = true;

}

if (currentToken.token == SingleLineCommentToken)

return NextToken();

return currentToken;

}

//Scanner Header File

#ifndef SCANNER\_H

#define SCANNER\_H

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

#include <iostream>

#include "tokens.h"

#include <sstream>

namespace patch {

template <typename T>

std::string to\_string(T);

}

class Scanner

{

private:

std::iostream \*input;

int lineNumber;

char next;

bool hasErrored;

TokenInfo currentToken;

void ReadEscapedCharacter();

void ParseString();

void ParseNumber();

void ConsumeWS();

void SkipNext();

public:

Scanner(std::iostream \*in)

{

input = in;

lineNumber = 1;

hasErrored = false;

}

bool HasToken();

bool HasErrored() { return hasErrored; }

TokenInfo NextToken();

char GetNext();

char GetFirst();

};

#endif

//ScannerTests

//Unit Test File

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

#include <iostream>

#include <sstream>

#include "scanner.h"

#include "QUnit.hpp"

using namespace std;

int main()

{

QUnit::UnitTest qunit(std::cerr, QUnit::normal);

stringstream ss;

Scanner s(&ss);

TokenInfo t;

int tmp;

for (int i = 1; i < NUM\_OF\_TOKENS; ++i) {

if (i == StringLitToken || i == CharLitToken || i == VoidToken || i == CharToken || i == StringToken || i == BoolToken)

continue;

ss.clear();

ss << lexeme((Token)i);

ss.flush();

cout << lexeme((Token)i) << " pushed on" << endl;

t = s.NextToken();

tmp = (int)t.token;

//cout << tmp << ", " <<i << endl;

QUNIT\_IS\_EQUAL(tmp, i);

cout << t.value << endl;

//cout << "remaining: " << ss<<endl;

cout << endl;

ss.str(std::string(""));

ss.clear();

}

return 0;

}

//SymbolTableTests

//Unit Test File

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

#include <iostream>

#include <sstream>

#include "QUnit.hpp"

#include "symTable.h"

#include "doggieTypes.h"

using namespace std;

int main()

{

QUnit::UnitTest qunit(std::cerr, QUnit::normal);

SymTable st;

const Record \*r;

QUNIT\_IS\_EQUAL(st.StackSize("a"), 0);

QUNIT\_IS\_FALSE(st.RecordExists("a"));

st.AddVariable("a", TypePoodle);

QUNIT\_IS\_TRUE(st.RecordExists("a"));

r = st.FindRecord("a");

QUNIT\_IS\_EQUAL(r->scope, 0);

st.AddFunction("b", TypePoodle);

st.AddVariable("a", TypePoodle);

st.AddVariable("c", TypePoodle);

st.AddVariable("d", TypePoodle);

st.AddVariable("e", TypePoodle);

st.AddVariable("f", TypePoodle);

r = st.FindRecord("a");

QUNIT\_IS\_EQUAL(r->scope, 1);

QUNIT\_IS\_FALSE(st.AddVariable("a", TypePoodle));

st.ReturnToPreviousScope();

cout << st.AlignedTableString() << endl;

st.AddVariable("d", TypePoodle);

st.AddFunction("c", TypePoodle);

st.AddParameter("param1", TypeDalmatian, false, 1);

cout << st.AlignedTableString() << endl;

return 0;

}

//symTable

//Group Symbol Table Project

// Jonathan Ringer

// Jeremy Vasseur

// Washing Kaleb

// Phillip Germagliotti

#include "symTable.h"

#include <string>

#include <iostream>

#include <memory>

using namespace std;

std::ostream& operator<<(std::ostream& os, Record r)

{

return os << r.name << " " << r.type << " " << r.scope;

}

template<typename T>

std::string to\_string( const T& n)

{

std::ostringstream stm;

stm << n;

return stm.str();

}

//push elements of a record size\_to an array, so we can align columns later

static vector<string> prsize\_tRecord(Record\* const r) {

vector<string> result;

result.push\_back(to\_string(r->name));

result.push\_back(to\_string(r->type));

result.push\_back(to\_string(r->scope));

result.push\_back(to\_string(r->size));

result.push\_back(to\_string(r->kind));

return result;

}

SymTable::SymTable()

{

currentScope = 0;

mostCurrentFuncNdx = 0;

}

SymTable::~SymTable() {};

size\_t SymTable:: StackSize(const string &name) const {

for (int i = this->records.size() - 1; i >= 0; i--) {

if (records[i].name == name && records[i].kind == KindFunc) {

//found record, now find variables declared at that scope

size\_t size = 0;

size\_t correctScope = records[i].scope + 1;

for (size\_t j = i + 1; j < records.size() && records[j].scope == correctScope; ++j)

size += records[j].size;

return size;

}

}

//nothing was found, return false

return 0;

}

bool SymTable::AddToStack(size\_t size)

{

//cout << "adding " << size << endl;

if (this->HasCurrent()) {

MostCurrentFunc().size += size;

return false;

}

return true;

}

bool SymTable::SetCurrentFuncsName( std:: string newName)

{

MostCurrentFunc().name = newName;

return false;

}

bool SymTable::AddArray(const std::string &name, enum Type type, size\_t num)

{

return AddRecord(name, type, KindVar, true, false, num);

}

bool SymTable::AddVariable(const std::string &name, enum Type type)

{

return AddRecord(name, type, KindVar, false, false, 1);

}

bool SymTable::AddParameter(const std::string &name, enum Type type, bool isArray, size\_t numOfElements) {

//cout << MostCurrentFunc().name << endl;

if (mostCurrentFuncNdx != 0)

return AddRecord(name, type, KindVar, isArray, true, numOfElements);

//most current scope was recently cleared... error

return false;

}

bool SymTable::RecordExists(const string &name, size\_t scope) const {

for (int i = this->records.size() - 1; i >= 0; i--) {

if (records[i].name == name && records[i].scope == scope)

return true;

}

//nothing was found, return false

return false;

}

//record name + scope is used to identify a given record

bool SymTable::RecordExists(const string &name) const {

//iterate through and see if we can't find something by the correct name and scope

for (int i = this->records.size() - 1; i >= 0; i--) {

if (records[i].name == name)

return true;

}

//nothing was found, return false

return false;

}

Record SymTable::FindRecord(const string &name) const

{

for (int i = records.size() - 1; i >= 0; --i)

if (records[i].name == name)

return records[i];

Record a;

//nothing was found, return default case

return a;

}

Record SymTable::FindRecord(const string &name, Kind kind) const

{

for (int i = records.size() - 1; i >= 0; i--)

if (records[i].name == name && records[i].kind == kind)

return records[i];

Record a;

//nothing was found, return default case

return a;

}

Record SymTable::FindRecord(const string &name, size\_t scope) const

{

for (int i = records.size() - 1; i >= 0; i--)

if (records[i].name == name && records[i].scope == scope)

return records[i];

Record a;

//nothing was found, return default case

return a;

}

//will return true if it successfully removed

//bool SymTable::RemoveLast() {

//

// if (this->IsEmpty())

// return false;

//

// Record r = records[records.size() - 1];

//

// //if we remove a function, we need to return to the previous scope

// if (r.Kind() == KindFunc)

// currentScope--;

// offsets[currentScope] -= r.Size();

//

// records.pop\_back();

//

// //nothing was found, return false

// return true;

//}

bool SymTable::IsEmpty() {

return records.size() < 1;

}

bool SymTable::AddFunction(const string &name, Type type)

{

return AddRecord(name, type, KindFunc, false, false, 1);

}

//returns true if it successfully added a record, returns false it a record already exists

bool SymTable::AddRecord(const string &name, Type type, Kind kind, bool isArray, bool isParameter, size\_t numOfElements = 1)

{

Record r;

//check to make sure that the id is unique

if (RecordExists(name, currentScope))

return false;

r.isArray = isArray && kind == KindVar;

r.isParameter = isParameter && kind == KindVar;

r.name = name;

r.type = type;

r.parentNdx = mostCurrentFuncNdx;

if (kind == KindVar){

r.size = TypeSize(type) \* numOfElements;

r.offset = this->CurrentOffset();

r.numOfElements = numOfElements;

AddToStack(r.size);

}

else {

r.size = 0;

r.offset = 0;

}

r.scope = currentScope;

r.kind = kind;

this->Add(r);

if (kind == KindFunc)

mostCurrentFuncNdx =records.size() -1;

//cout << "current:" << MostCurrentFunc().name << endl;

//id is unique, add to table

//increment scope if it was a function

if (kind == KindFunc) {

this->IncrementScope();

}

//if it was a variable, increase stack size of function

//cout << this->AlignedTableString() << endl;

return true;

}

size\_t SymTable::CurrentOffset()

{

//if there is no current entry, then start at 0

if (offsets.size() == currentScope)

offsets.push\_back(0);

//sanity check

if (offsets.size() >= (size\_t)currentScope)

return this->offsets[currentScope];

return 0;

}

void SymTable::Add(Record r)

{

records.push\_back(r);

//sanity check

if (offsets.size() <= (size\_t)currentScope)

offsets.push\_back(0);

//increment offset for next element

if (offsets.size() >= (size\_t)currentScope)

offsets[currentScope] += r.size;

}

//deletes all records at the top level scope

void SymTable::ReturnToPreviousScope()

{

//cannot return from topmost level scope

if (currentScope <= 0)

return;

//remove all records of the scope we are leaving

for (int i = records.size() - 1; i >= 0; i--)

if (records[i].scope == currentScope)

records.pop\_back();

this->DecrementScope();

//TODO most current func needs to be set to parent scope

}

bool SymTable::SetCurrentFuncsLocation(size\_t location)

{

MostCurrentFunc().size = location;

return true;

}

size\_t SymTable::CurrentScope()

{

return currentScope;

}

size\_t SymTable::MostCurrentStackSize()

{

return MostCurrentFunc().size;

}

void SymTable::IncrementScope()

{

++currentScope;

//make sure it has an offset value

if (offsets.size() == currentScope)

offsets.push\_back(0);

else

offsets[currentScope] = 0;

}

//handles removal of a scope

void SymTable::DecrementScope()

{

if (records.size() > 0 && currentScope > 0)

offsets[currentScope--] = 0;

if (HasCurrent())

mostCurrentFuncNdx = MostCurrentFunc().parentNdx;

}

//returns record table with aligned columns

std::string SymTable::AlignedTableString()

{

vector< vector<string> > cells;

size\_t widths[8]; //widths of the columns

//header

vector<string> header;

header.push\_back("Name");

header.push\_back("Type");

header.push\_back("Scope");

header.push\_back("Size");

header.push\_back("Kind");

cells.push\_back(header);

//split records size\_to cells

for (vector<Record>::iterator it = records.begin(); it != records.end(); it++)

cells.push\_back(prsize\_tRecord(&(\*(it))));

//find the greatest width in each column

for (size\_t i = 0; i < RECORD\_ELEMENTS; ++i) {

size\_t w = 0;

for (size\_t j = 0; j < cells.size(); ++j) {

size\_t a =cells[j][i].length();

w = a > w ? a : w;

}

widths[i] = w;

}

//set all cells in a given column to the same width

for(size\_t i = 0; i < RECORD\_ELEMENTS; ++i)

for (size\_t j = 0; j < cells.size(); ++j)

cells[j][i].resize(widths[i],' ');

//concat all the rows

stringstream ss;

for (size\_t i = 0; i < cells.size(); ++i) {

for(size\_t j = 0; j < cells[i].size(); ++j)

ss << cells[i][j] << " ";

ss << endl;

}

return ss.str();

}

//SymTable Header

#ifndef SYMTABLE\_H

#define SYMTABLE\_H

//Group Symbol Table Project

// Jonathan Ringer

// Jeremy Vasseur

// Washing Kaleb

// Phillip Germagliotti

#include <vector>

#include <iostream>

#include <sstream>

#include <string>

#include <stack>

#include "doggieTypes.h"

//patch for c++11 equivalent function

template<typename T>

std::string to\_string( const T& n);

/\*

template<typename T>

struct optional {

T value;

bool hasValue;

} optional;

\*/

template<typename T>

std::string to\_string( const T& n);

struct Parameter {

public:

std::string name;

enum Type type;

bool isArray;

size\_t numOfElements;

};

struct Record {

public:

std::string name;

Type type;

size\_t scope;

size\_t parentNdx;

enum Kind kind;

size\_t size;

size\_t line\_addr; //function location

size\_t offset; //local offset, for variable

bool isParameter;

bool isArray;

size\_t numOfElements;

};

//std::ostream& operator<<(std::ostream& os, Record r);

#define RECORD\_ELEMENTS 5

//END Type definitions

class SymTable

{

private:

std::vector<Record> records; //"table" of records

std::vector<size\_t> offsets; //will give an offset for a given scope level as the index

size\_t currentScope; //determines which scope we append new records to

void DecrementScope(); //used to handle offset information when leaving a scope

void IncrementScope(); //inserts a new offset value

void Add(Record r);

size\_t CurrentOffset();

size\_t mostCurrentFuncNdx;

Record& MostCurrentFunc() {return records[mostCurrentFuncNdx]; }

bool AddRecord(const std::string &name, Type type, Kind kind, bool isArray, bool isParameter, size\_t numOfElems);

public:

SymTable();

~SymTable();

bool IsEmpty();

bool AddToStack(size\_t);

bool HasCurrent() { return mostCurrentFuncNdx != 0;}

size\_t CurrentScope();

//returns true if it successfully added a record, returns false it a record already exist

bool AddFunction(const std::string &name, Type type);

bool SetCurrentFuncsName( std:: string);

bool SetCurrentFuncsLocation(size\_t location);

bool AddVariable(const std::string &name, Type type);

//add an array

bool AddArray(const std::string &name, Type type, size\_t num);

//Adds a parameter to the most current declared function, will error if still in global scope

bool AddParameter(const std::string &name, Type type, bool, size\_t);

//finds the most recent function with the given name, then returns the size of all parameters and local variables.

//returns -1 if records could not be found

size\_t StackSize(const std::string &name) const;

size\_t MostCurrentStackSize();

//returns the number of parameters for the most recent function

size\_t NumOfParameters(const std::string &name);

//record name + scope is used to identify a given record

bool RecordExists(const std::string &name) const;

bool RecordExists(const std::string &name, size\_t scope) const;

Record FindRecord(const std::string &name) const;

Record FindRecord(const std::string &name, Kind kind) const;

Record FindRecord(const std::string &name, size\_t scope) const;

//will return true if it successfully removed

//not sure where this method to be useful, because individual records wouldnt fall out of scope.... oh well

//bool RemoveLast();

//deletes all records at the top level scope

void ReturnToPreviousScope();

//returns record table with aligned columns

std::string AlignedTableString();

};

#endif

//Tokens

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

#include <iostream>

#include "tokens.h"

std::string lexeme(enum Token token)

{

if (token == EoLToken) return "!";

else if (token == AddToken) return "#";

else if (token == SubToken) return "\_/";

else if (token == MultToken) return "<3";

else if (token == DivToken) return "VV";

else if (token == ModToken) return "{}";

else if (token == IncrToken) return "/T";

else if (token == DecrToken) return "\\T";

else if (token == AssignToken) return "/|";

else if (token == LessToken) return "<}";

else if (token == GreaterToken) return "{>";

else if (token == EqToken) return " <>";

else if (token == NegateToken) return ":3";

else if (token == AndToken) return "N'";

else if (token == OrToken) return "//";

else if (token == TrueToken) return "OO";

else if (token == FalseToken) return "XX";

else if (token == HexToken) return "@";

else if (token == CharLitToken) return "^";

else if (token == StringLitToken) return "$";

else if (token == StringEscToken) return "\*";

else if (token == VoidDecToken) return " Stray";

else if (token == BoolDecToken) return " Dalmatian";

else if (token == IntDecToken) return " Poodle";

else if (token == CharDecToken) return " Yorkie";

else if (token == RealDecToken) return " Corgi";

/\*else if (token == StringDecToken) return " Dachshund";\*/

else if (token == NameToken) return " someName";

else if (token == VoidToken) return " Stray";

else if (token == BoolToken) return " OO";

else if (token == IntToken) return "232";

else if (token == CharToken) return "^d^";

else if (token == RealToken) return "5.2";

else if (token == StringToken) return "$d$";

else if (token == LParenToken) return "@=";

else if (token == RParenToken) return "=@";

else if (token == SeparatorToken) return "::";

else if (token == ArrayDefToken) return "|";

else if (token == PointerDefToken) return "..";

else if (token == DerefToken) return "--";

else if (token == SingleLineCommentToken) return " PANT\n";

else if (token == LMultiLineCommentToken) return " GRR";

else if (token == RMultiLineCommentToken) return " BARK";

else if (token == LBlockToken) return " SQUIRREL";

else if (token == RBlockToken) return " Nevermind...";

else if (token == IfToken) return " Sniff";

else if (token == ElseToken) return " Sneeze";

else if (token == SwitchToken) return " NAPTIME";

else if (token == CaseToken) return " Bed";

else if (token == DefaultToken) return " Doghouse";

else if (token == BreakToken) return " Wake!";

else if (token == WhileToken) return " Chase";

else if (token == ForToken) return " Rollover";

else if (token == FromToken) return " From";

else if (token == ToToken) return " To";

else if (token == RepeatToken) return " Scratch";

else if (token == UntilToken) return " NOFLEAS";

else if (token == InputIntToken) return " FeedPoodle";

else if (token == InputCharToken) return " FeedYorkie";

else if (token == InputRealToken) return " FeedCorgi";

/\*else if (token == InputStringToken) return " FeedDachshund";\*/

else if (token == InputBoolToken) return " FeedDalmatian";

else if (token == PrintIntToken) return " SpeakPoodle";

else if (token == PrintCharToken) return " SpeakYorkie";

else if (token == PrintRealToken) return " SpeakCorgi";

/\*else if (token == PrintStringToken) return " SpeakDachshund";\*/

else if (token == PrintBoolToken) return " SpeakDalmation";

else if (token == LessOrEqualToken) return "<}<>";

else if (token == GreaterOrEqualToken) return "{><>";

else if (token == NotEqualToken) return ":3<>";

else if (token == PositiveToken) return "+&";

else if (token == NegativeToken) return "-&";

else if (token == BedEndToken) return "?";

else if (token == AddressOfToken) return "|P";

else if (token == ErrorToken) return "ErrorToken";

return "Error";

}

std::ostream& operator<<(std::ostream& os, enum Token token)

{

if (token == EoLToken) return os << "EoLToken";

else if (token == AddToken) return os << "AddToken";

else if (token == SubToken) return os << "SubToken";

else if (token == MultToken) return os << "MultToken";

else if (token == DivToken) return os << "DivToken";

else if (token == ModToken) return os << "ModToken";

else if (token == IncrToken) return os << "IncrToken";

else if (token == DecrToken) return os << "DecrToken";

else if (token == AssignToken) return os << "AssignToken";

else if (token == LessToken) return os << "LessToken";

else if (token == GreaterToken) return os << "GreaterToken";

else if (token == EqToken) return os << "EqToken";

else if (token == NegateToken) return os << "NegateToken";

else if (token == AndToken) return os << "AndToken";

else if (token == OrToken) return os << "OrToken";

else if (token == TrueToken) return os << "TrueToken";

else if (token == FalseToken) return os << "FalseToken";

else if (token == HexToken) return os << "HexToken";

else if (token == CharLitToken) return os << "CharLitToken";

else if (token == StringLitToken) return os << "StringLitToken";

else if (token == StringEscToken) return os << "StringEscToken";

else if (token == VoidDecToken) return os << "VoidDecToken";

else if (token == BoolDecToken) return os << "BoolDecToken";

else if (token == IntDecToken) return os << "IntDecToken";

else if (token == CharDecToken) return os << "CharDecToken";

else if (token == RealDecToken) return os << "RealDecToken";

/\* else if (token == StringDecToken) return os << "StringDecToken";\*/

else if (token == NameToken) return os << "NameToken";

else if (token == VoidToken) return os << "VoidToken";

else if (token == BoolToken) return os << "BoolToken";

else if (token == IntToken) return os << "IntToken";

else if (token == CharToken) return os << "CharToken";

else if (token == RealToken) return os << "RealToken";

else if (token == StringToken) return os << "StringToken";

else if (token == LParenToken) return os << "LParenToken";

else if (token == RParenToken) return os << "RParenToken";

else if (token == SeparatorToken) return os << "SeperatorToken";

else if (token == ArrayDefToken) return os << "ArrayDefToken";

else if (token == PointerDefToken) return os << "PointerDefToken";

else if (token == DerefToken) return os << "DerefToken";

else if (token == SingleLineCommentToken) return os << "SingleLineCommentToken";

else if (token == LMultiLineCommentToken) return os << "LMultiLineCommentToken";

else if (token == RMultiLineCommentToken) return os << "RMultiLineCommentToken";

else if (token == LBlockToken) return os << "LBlockToken";

else if (token == RBlockToken) return os << "RBlockToken";

else if (token == IfToken) return os << "IfToken";

else if (token == ElseToken) return os << "ElseToken";

else if (token == SwitchToken) return os << "SwitchToken";

else if (token == CaseToken) return os << "CaseToken";

else if (token == WhileToken) return os << "WhileToken";

else if (token == ForToken) return os << "ForToken";

else if (token == FromToken) return os << "FromToken";

else if (token == ToToken) return os << "ToToken";

else if (token == RepeatToken) return os << "RepeatToken";

else if (token == UntilToken) return os << "UntilToken";

else if (token == InputIntToken) return os << "InputIntToken";

else if (token == InputCharToken) return os << "InputCharToken";

else if (token == InputRealToken) return os << "InputRealToken";

/\* else if (token == InputStringToken) return os << "InputStringToken";\*/

else if (token == InputBoolToken) return os << "InputBoolToken";

else if (token == PrintIntToken) return os << "PrintIntToken";

else if (token == PrintCharToken) return os << "PrintCharToken";

else if (token == PrintRealToken) return os << "PrintRealToken";

/\* else if (token == PrintStringToken) return os << "PrintStringToken";\*/

else if (token == PrintBoolToken) return os << "PrintBoolToken";

else if (token == LessOrEqualToken) return os << "LessOrEqualToken";

else if (token == GreaterOrEqualToken) return os << "GreaterOrEqualToken";

else if (token == NotEqualToken) return os << " NotEqualToken";

else if (token == PositiveToken) return os << "PositiveToken";

else if (token == NegativeToken) return os << "NegativeToken";

else if (token == BedEndToken) return os << "BedEndToken";

else if (token == AddressOfToken) return os << "AddressOfToken";

else if (token == ErrorToken) return os << "ErrorToken";

return os << "Error";

}

std::ostream& operator<<(std::ostream& os, TokenInfo token)

{

return os << "{ token: " << token.token << " lex: " << token.value.c\_str() << " }";

}

//Tokens Header

#ifndef TOKENS\_H

#define TOKENS\_H

//Jonathan Ringer

//Kaleb Washington

//Jeremy Vasseur

//Phillip Germagliotti

#include <iostream>

// This file defines the tokens that are available in the doggiestyle language

#define NUM\_OF\_TOKENS 70

//C-keyword Doggie

enum Token { EoLToken = 1, //; !

LessOrEqualToken, //<= <}<>

GreaterOrEqualToken, //>= {><>

NotEqualToken, //!= :3<>

PositiveToken, //+num +&

NegativeToken, //-num -&

BedEndToken, // ?

AddressOfToken, // |P

AddToken, //+ #

SubToken, //- \_/

MultToken, //\* <3

DivToken, /// VV

ModToken, //% {}

IncrToken, //++ /T

DecrToken, //-- \T

AssignToken, //= /|

LessToken, //< <}

GreaterToken, //> {>

EqToken, //== <>

NegateToken, //! :3

AndToken, //&& N'

OrToken, //|| //

TrueToken, //true oo

FalseToken, //false xx

HexToken, //0x @

CharLitToken, //' ^

StringLitToken, //" $

StringEscToken, //\ \*

VoidDecToken, //void Stray

BoolDecToken, //bool Dalmatian

IntDecToken, //int Poodle

CharDecToken, //char Yorkie

RealDecToken, //double Corgi

/\* StringDecToken,\*/ //string Dachshund

NameToken, //<id> <id>

VoidToken, //???? ????

BoolToken, //true/false oo/xx

IntToken, //52 52

CharToken, //'c' ^c^

RealToken, //52.1 52.1

StringToken, //"hello" $hello$

LParenToken, //( 8=

RParenToken, //) =8

SeparatorToken, //, ::

ArrayDefToken, //[ |

PointerDefToken, //\* ..

DerefToken, //\* --

SingleLineCommentToken,//// PANT

LMultiLineCommentToken,///\* GRR

RMultiLineCommentToken,//\*/ BARK

LBlockToken, //{ SQUIRREL

RBlockToken, //} NEVERMIND...

IfToken, //if SNIFF

ElseToken, //else SNEEZE

SwitchToken, //switch NAPTIME

CaseToken, //case BED

BreakToken, //break Wake

DefaultToken, //default Doghouse

WhileToken, //while CHASE

ForToken, //for ROLLOVER

FromToken, // ---- FROM

ToToken, // ---- TO

RepeatToken, // ---- SCRATCH

UntilToken, // ---- NOFLEAS

InputIntToken, // --- FEEDPOODLE

InputCharToken, // --- FEEDYORKIE

InputRealToken, // ---- FEEDCORGI

/\* InputStringToken,\*/ // ---- FEEDDACHSHUND

InputBoolToken, // ---- FEEDDALMATION

PrintIntToken, // --- SPEAKPOODLE

PrintCharToken, // --- SPEAKYORKIE

PrintRealToken, // ---- SPEAKCORGI

/\*PrintStringToken, \*/ // ---- SPEAKDACHSHUND

PrintBoolToken, // ---- SPEAKDALMATION

ErrorToken, //parse error

FetchToken, //----- ----

};

typedef struct {

Token token;

std::string value;

} TokenInfo;

std::string lexeme(enum Token);

std::ostream& operator<<(std::ostream& os, enum Token);

std::ostream& operator<<(std::ostream& os, TokenInfo token);

#endif