1 compile.c

```c
/*
 * compile.c
 *
 *  Created on: 12 Oct 2016
 *      Author: harry
 */

#define _GNU_SOURCE
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include "TacLineQueue.h"
#include "MIPSMemoryInfo.h"


typedef struct TacLine TacLine;

//keeps track of what temperories are avaliable
int temp_count = 0;
//keeps track of what if statement or while statement the compiler is compiling
int label_count = 0;

char* MAINSTRING = "main";
char* IFSTRING = "IF";

void printVarAssignment(char* variable, char* variable2,int op);
void createVarAssignment(char* variable, char* variable2,int op,int isVar1Temp,
int isVar2Temp,
    int isVariableCreation);
struct TypeValue IfVarWrap(struct TypeValue var,int isVariableCreation);
void printParam(struct TypeValue value);
void createParam(struct TypeValue value);
void printPopArg(struct TypeValue value);
void createPopArg(struct TypeValue value);
struct TypeValue placeInterInTemp(struct TypeValue value);
void createFunctionCall(char* function,int temp,int applyOnTemp);
void createStatement0(char* statement,int isFunction,char operator);
void replaceIfReserved(char** functionName);
struct TypeValue compile0(NODE* tree,int variableCreated);
void createIfCode(NODE* tree, int variableCreated,int fromElse);
void createStatement(char* statement);
void createSimpleInstruct(char* variable,int operand1,int isVar1,int op);
void createInstruction(char* variable,int operand1,int isVar1,
int operand2,int isVar2, int operator);
void createInstruction0(char* variable,int operand1,int isVar1,
int operand2,int isVar2, int operator,
    int thereIsElse);
void printSimpleAssignment(char* var, int value, int isVar);
void printTacLine(char* var,int op, struct TypeValue left, struct TypeValue right);
void printOperand(struct TypeValue operand,int isRight);
int genTemp();
void resetTemp();
int genLabelCount();
int getLabelCount();
void releaseLabelCount();
int reuseTemp();
int strNumSize(int value,int strLength);
void intToString(char* start,int value,char* end,char** combine);
int numDigits(int number);
void compile(NODE* tree);


struct TypeValue{

  int value;
  char* lexeme;
  int type;
};

//prints a simple variable assignment
void printVarAssignment(char* variable, char* variable2,int op){

  printf("%s %c %s;\n",variable,op,variable2);

}

//creates a TAC line structure for a simple variable assignment
//is variable creation is 1 when this variable is a creation and not an update
void createVarAssignment(char* variable, char* variable2,int op,int isVar1Temp,
int isVar2Temp,
    int isVariableCreation){

  TacLine* tacline = (TacLine*)malloc(sizeof(TacLine));
  tacline->variable = variable;
  tacline->isVar1Temp = isVar1Temp;
  tacline->isVar2Temp = isVar2Temp;
  tacline->variable2 = variable2;
  tacline->operator = op;
  tacline->isNext = 0;
  tacline->isStatement = 0;
  tacline->isVariableEq = 1;
  tacline->isSimple = 1;
  tacline->isRegisterFunctionCall = 0;
  tacline->isVariableCreation = isVariableCreation;
  tacline->next = NULL;

  addToQueue(tacline);
}

//places a variable inside a temporary
struct TypeValue IfVarWrap(struct TypeValue var,int isVariableCreation){

  if(var.type == 3){

    int d = genTemp();
    char* temp;
    intToString("$t",d,"",&temp);
    printVarAssignment(temp,var.lexeme,'=');
    createVarAssignment(temp,var.lexeme,'=',1,0,isVariableCreation);

    struct TypeValue new;
    new.value = d;
    new.type = 1;
    return new;

  }else{
    return var;
  }
}

//prints a parameter that is being pushed
void printParam(struct TypeValue value){

  if(value.type == 1){

    printf("PushParam␣$t%d;\n",value.value);

  }else if(value.type == 3){

    printf("PushParam␣%s;\n",value.lexeme);
  }
}

//creates a TAC lien that represents a parameter being pushed
void createParam(struct TypeValue value){

  TacLine* tacline = (TacLine*)malloc(sizeof(TacLine));
  tacline->operator = 'P';
  tacline->isNext = 0;
  tacline->isSimple = 0;
  tacline->isStatement = 0;
  tacline->isVariableEq = 0;
  tacline->variable2 = NULL;
  tacline->isRegisterFunctionCall = 0;
  tacline->paramType = value.type;

  //if a integer is being pushed
  if(value.type == 1){

    tacline->operand1 = value.value;
    tacline->isVar1 = 1;

  //if a variable is being pushed
  }else if(value.type == 3){

    tacline->variable = value.lexeme;
  }

  addToQueue(tacline);
}

//prints an argument being popped when a function has been called
void printPopArg(struct TypeValue value){

  printf("PopArg␣%s;\n",value.lexeme);

}

//creates a Tac line representing a argument being placed in the function's frame
void createPopArg(struct TypeValue value){

  TacLine* line = (TacLine*)malloc(sizeof(TacLine));
  line->operator = 'A';
  line->variable = value.lexeme;
  line->paramType = value.type;
  line->variable2 = NULL;
  line->isRegisterFunctionCall = 0;

  addToQueue(line);
}

//this function is used to get around the limitations in MIPS where the first operand of a opera
//has to be a register
struct TypeValue placeInterInTemp(struct TypeValue value){

  //if the value is an integer then place that integer into a temporary
  if(value.type == 0){

    int temp = reuseTemp();
    char* var;
    intToString("$t",temp,"",&var);
    printSimpleAssignment(var,value.value,0);
    createSimpleInstruct(var,value.value,0,'=');

    struct TypeValue newValue;
    newValue.type = 1;
    newValue.value = temp;
    return newValue;

  //if the value is variable place it in a temporary
  }else if(value.type == 3){

    int temp = reuseTemp();
    char* var;
    intToString("$t",temp,"",&var);
    printVarAssignment(var,value.lexeme,'=');
    createVarAssignment(var,value.lexeme,'=',1,0,0);
    struct TypeValue newValue;
    newValue.type = 1;
    newValue.value = temp;
    return newValue;

  }else{

    return value;
  }
}

//this function creates a tac line representing a function call
//1st arg: function name 2nd arg: the temporary to place the returned value
//3rd arg: whether this function call is on a function pointer
void createFunctionCall(char* function,int temp,int applyOnTemp){

  TacLine* line = (TacLine*)malloc(sizeof(TacLine));
  line->variable = function;
  line->variable2 = NULL;
  line->operator = 'F';
  line->isStatement = 0;
  line->isNext = 0;
  line->isVar1Temp = 1;
  line->operand1 = temp;
  line->isSimple = 0;
  line->isRegisterFunctionCall = applyOnTemp;
  line->next = NULL;
  addToQueue(line);
}

//creates a generic function
//1st: the name of the statement 2nd arg: whether this is a function declaration
//3rd: the statement's type
void createStatement0(char* statement,int isFunction,char operator){

  TacLine* line = (TacLine*)malloc(sizeof(TacLine));
  line->variable = statement;
  line->variable2 = NULL;
  line->isStatement = 1 + isFunction;
  line->isVariableEq = 0;
  line->isNext = 0;
  line->operator = operator;
  line->isRegisterFunctionCall = 0;
  line->next = NULL;

  addToQueue(line);
}

//adds a _ to the end of function names that have the same name as MIPS
reserved words
void replaceIfReserved(char** functionName){

  const char *a[4];
  a[0] = "add";
  a[1] = "mul";
  a[2] = "sub";
  a[3] = "div";

  int i;
  for(i = 0; i < 4; i++){

    //if the function name is the same as the reserved work then add the
    letter _
    if(strcmp(*functionName,a[i]) == 0){

      char* oldWord = *functionName;
      *functionName = (char*)malloc(sizeof(strlen(*functionName)+2));
      strcpy(*functionName,oldWord);
      strcat(*functionName,"_");
    }
  }
}

//inital compilation phase
//1st arg: the next node to compile 2nd: whether the next assignment is a
creation or update
struct TypeValue compile0(NODE* tree,int variableCreated){

  //the node type determines how a node is compiled
  switch(tree->type){

  //new function declaration
  case 'D':

    //can start using temporaries from 0 again as this is a new function
    resetTemp();
    TOKEN* name = (TOKEN*)tree->left->right->left->left;
    //replace the function name if it's a reserved work in MIPS
    replaceIfReserved(&name->lexeme);
    printf("%s:␣\n",name->lexeme);
    //create the TACLINE
    createStatement(name->lexeme);
    //compile the function
    compile0(tree->left,0);
    compile0(tree->right,0);
    //end the function with .end function name
    char* start = ".end␣";
    char* endState = (char*)malloc((strlen(start) +
    strlen(name->lexeme) + 1) * sizeof(char));
    strcpy(endState,start);

    strcat(endState,name->lexeme);
    printf("%s\n",endState);
    //create end function statement
    createStatement0(endState,3,'E');
    break;

  //start compiling the arguments by printing the only argument or
  recursiving on the , node
  //to find the other arguments
  case 'F':

    compile0(tree->left,variableCreated);
    if(tree->right != NULL){
      if(tree->right->type == ','){

        compile0(tree->right,variableCreated);

      }else{

        struct TypeValue arg = compile0(tree->right->right,
        variableCreated);
        printPopArg(arg);
        createPopArg(arg);
      }
    }

    break;
  //comma indicates either multiple arguments or multiple parameters
  case ',':

    ;
    if(tree->left->type != ','){
      //~ indicates that it's a argument being passed to a function
      if(tree->left->type == '~'){

        struct TypeValue valueL = compile0(tree->left->right,
        variableCreated);
        printPopArg(valueL);
        createPopArg(valueL);

      //otherwise it's a parameter being parsed by a function
      }else{

        struct TypeValue valueL = compile0(tree->left,variableCreated);

        valueL = placeInterInTemp(valueL);
        printParam(valueL);
        createParam(valueL);
      }
    }else{

      //if the next type is a comma than keep looking
      compile0(tree->left,variableCreated);
    }

    //same but with the right node
    if(tree->right->type != ','){
      if(tree->right->type == '~'){

        struct TypeValue valueR = compile0(tree->right->right,
        variableCreated);
        printPopArg(valueR);
        createPopArg(valueR);

      }else{

        struct TypeValue valueR = compile0(tree->right,variableCreated);
        valueR = placeInterInTemp(valueR);
        printParam(valueR);
        createParam(valueR);
      }
    }else{

      compile0(tree->right,variableCreated);
    }
    break;

  //if a function is being called
  case APPLY:

    ;
    struct TypeValue onApply;
    int applyOnFunction = 0;
    //a nested function call (on returned function pointer)
    if(tree->left->type == APPLY){

      //get result before call this function
      onApply = compile0(tree->left,variableCreated);
      applyOnFunction = 1;
    }

    genTemp();
    //find the parameters
    if(tree->right != NULL){
      if(tree->right->type == ','){

        compile0(tree->right,variableCreated);

      }else{

        struct TypeValue param = compile0(tree->right,variableCreated);

        param = placeInterInTemp(param);
        printParam(param);
        createParam(param);
      }
```

```
      }

      TOKEN* call = (TOKEN*)tree->left->left;

      //get the function to call
      char* functionName;
      if(!applyOnFunction){

        functionName = call->lexeme;

      }else{

        //get the address of the function from the temporary
        intToString("$t",onApply.value,"",&functionName);
      }
      //get a temporary to place the return value in
      int temp1 = genTemp();
      printf("$t%d␣=␣LCall␣%s;\n",temp1,functionName);
      //create the function call TacLine
      createFunctionCall(functionName,temp1,applyOnFunction);

      //return the function return value
      struct TypeValue funcBack;
      funcBack.type = 1;
      funcBack.value = temp1;

      return funcBack;

    //if this node is a return node
    case RETURN:
      {
        struct TypeValue value = compile0(tree->left,variableCreated);

        char* start = "Return␣";

        //create the return node
        TacLine* line = (TacLine*)malloc(sizeof(TacLine));
        line->operator = 'R';
        line->isRegisterFunctionCall = 0;
        line->variable2 = NULL;

        //type = 0 is integer
        if(value.type == 0){

          printf("%s␣%d;\n",start,value.value);
          line->operand1 = value.value;
          line->isVar1 = 0;


        //type = 1 is temporary
        }else if(value.type == 1){

          printf("%s␣$t%d;\n",start,value.value);
          line->operand1 = value.value;
          line->isVar1Temp = 1;
          line->isVar1 = 0;

        //type = 3 is variable
        }else if(value.type == 3){
```

```
      printf("%s␣%s;\n",start,value.lexeme);
      line->variable = value.lexeme;
      line->isVar1 = 1;
    }

    line->next = NULL;
    addToQueue(line);
    line = NULL;

  }

  break;

//if the node is an assignment
case '=':

  ;
  //compile the left side of the assignment
  struct TypeValue a = compile0(tree->right,variableCreated);

  int t = genTemp();
  char* temp;
  intToString("$t",t,"",&temp);
  //place the value of the left side into a temporary
  printSimpleAssignment(temp,a.value,a.type == 1);
  createSimpleInstruct(temp,a.value,a.type == 1,'=');
  TOKEN* variable = ((TOKEN*)tree->left->left);
  printVarAssignment(variable->lexeme,temp,'=');
  createVarAssignment(variable->lexeme,temp,'=',0,1,variableCreated);

  break;

//if a node is an operator on two operands
case '+':
case '*':
case '-':
case '/':
case '<':
case '>':
case EQ_OP:
  ;
  //break down the left and right side
  struct TypeValue l = compile0(tree->left,variableCreated);
  struct TypeValue r = compile0(tree->right,variableCreated);

  //in l and r is either a value or a temporary
  //place whats in l and r into temporary
  int d = genTemp();
  char* var;
  intToString("$t",d,"",&var);

  l = IfVarWrap(l,variableCreated);
  r = IfVarWrap(r,variableCreated);

  if(l.type == 0 && r.type == 0){

    int d2 = genTemp();
    char* inter;
    intToString("$t",d2,"",&inter);
```

```
        createSimpleInstruct(inter,l.value,0,'=');
        printSimpleAssignment(inter,l.value,0);

        struct TypeValue lv;
        lv.value = d2;
        lv.type = 1;

        printTacLine(var,tree->type,lv,r);
        createInstruction(var,lv.value,lv.type,r.value,r.type,tree->type);

    }else{

        printTacLine(var,tree->type,l,r);
        createInstruction(var,l.value,l.type,r.value,r.type,tree->type);
    }

    struct TypeValue tv;
    tv.value = d;
    tv.type = 1;
    return tv;

//if statement
case IF:
    ;

    //parse the condition
    struct TypeValue condition = compile0(tree->left,variableCreated);
    if(condition.type == 0){

        printf("IF␣%d:\n",condition.value);

    }else if(condition.type == 1){

        printf("IF␣$t%d:\n",condition.value);

    }else{

        printf("IF␣%s:\n",condition.lexeme);
    }


    //reset temp count as we are in a new context
    resetTemp();
    //create if label to then code
    char* ifLabel;
    int labelC = genLabelCount();
    asprintf(&ifLabel,"if␣%d",labelC);
    if(condition.type == 0){

        condition = IfVarWrap(condition,0);
    }
    createInstruction0(ifLabel,condition.value,condition.type == 1,1,0,'C',
        tree->right->type == ELSE);

    //create final pieces of if code it there is no else statement
    if(tree->right->type != ELSE){

        createIfCode(tree,variableCreated,0);

    }
    //create if block
    compile0(tree->right,variableCreated);


    //create label to end of if statement
    //else statement will do it if it exists
    if(tree->right->type != ELSE){
      char* endIf;
      asprintf(&endIf,"end_%d",getLabelCount());
      createStatement0(endIf,0,'J');
      printf("END_THEN\n");
    }

    //stops nested ifs from mislabeling
    releaseLabelCount();

    break;

  case ELSE:

    resetTemp();
    //create else block
    printf("ELSE: \n");
    compile0(tree->right,variableCreated);
    printf("END_ELSE\n");
    createIfCode(tree,variableCreated,1);
    printf("END_THEN\n");

    break;

  //create while loop
  case WHILE:

    ;
    char* endWhileLabel;
    //create while label
    int whileLabelCount = genLabelCount();
    asprintf(&endWhileLabel,"While_%d",whileLabelCount);
    createStatement0(endWhileLabel,0,'B');
    //create while condition
    struct TypeValue whileCondition = compile0(tree->left,variableCreated);

    //print while statement
    if(whileCondition.type == 0){

      printf("WHILE_%d:\n",whileCondition.value);

    }else if(whileCondition.type == 1){

      printf("WHILE_$t%d:\n",whileCondition.value);

    }else{

      printf("WHILE_%s:\n",whileCondition.lexeme);
    }

    //create end while label
    char* whileLabel;
    asprintf(&whileLabel,"endWhile_%d",whileLabelCount);
    //if while (integer) is encountered
    //then place integer in temporary
    if(whileCondition.type == 0){

      int temp = genTemp();
      char* inter;
      intToString("$t",temp,"",&inter);
      printSimpleAssignment(inter,whileCondition.value,0);
      createSimpleInstruct(inter,whileCondition.value,0,'=');
      createInstruction0(whileLabel,temp,1,0,0,'W',0);
    }else{

      createInstruction0(whileLabel,whileCondition.value,
      whileCondition.type == 1,0,0,'W',0);
    }


    //create while block
    resetTemp();
    compile0(tree->right,variableCreated);

    //printf end while label
    char* jumpWhileLabel;
    asprintf(&jumpWhileLabel,"j␣While_%d",whileLabelCount);
    createStatement0(jumpWhileLabel,0,'M');

    createStatement0(whileLabel,0,'B');
    printf("END_WHILE\n");
    releaseLabelCount();

    break;
  case LEAF:

    //if this node is a leaf
    //then it's either a constant or an identifier
    if(tree->left->type == CONSTANT){
      struct TypeValue v;
      v.value = ((TOKEN*)tree->left)->value;
      v.type = 0;
      return v;

    }else if(tree->left->type == IDENTIFIER){

      struct TypeValue a;
      a.lexeme = ((TOKEN*)tree->left)->lexeme;
      a.type = 3;
      return a;

    }
    break;

  case '~':

    //proceeding this is a creation of a variable
    //regardless of similar variables in parent functions
    compile0(tree->left,1);
    if(tree->right != NULL){
      compile0(tree->right,1);
    }
    break;
  default:

    //keep looking
    compile0(tree->left, variableCreated);
    if(tree->right != NULL){
      compile0(tree->right, variableCreated);
    }
  }

  //return nothing
  struct TypeValue n;
  n.value = 0;
  n.type = 2;
  return n;
}

void createIfCode(NODE* tree, int variableCreated, int fromElse){

  char* endIf;
  //create end of else jump
  asprintf(&endIf,"end_%d",getLabelCount());
  char* jumpToEnd = (char*)malloc(sizeof(char)*(strlen(endIf) + 5));
  strcpy(jumpToEnd,"j␣");
  strcat(jumpToEnd,endIf);
  char jumpToEndOp = 'O';
  if(fromElse){
    jumpToEndOp = 'M';
  }
  createStatement0(jumpToEnd,0,jumpToEndOp);
  //create jump to if code
  printf("THEN:␣\n");
  char* ifLabel2;
  asprintf(&ifLabel2,"if_%d",getLabelCount());
  createStatement0(ifLabel2,0,'K');
  if(!fromElse){

    compile0(tree->left->left, variableCreated);

  }else{

    compile0(tree->left, variableCreated);
    createStatement0(endIf,0,'J');
  }

}

void createStatement(char* statement){

  createStatement0(statement,1,'D');
}

void createSimpleInstruct(char* variable,int operand1,int isVar1,int op){

  TacLine* line = (TacLine*)malloc(sizeof(TacLine));
  line->variable = variable;
  line->operand1 = operand1;
  line->isVar1 = isVar1;
  line->isSimple = 1;
  line->isStatement = 0;
  line->isVariableEq = 0;
  line->isRegisterFunctionCall = 0;
  line->operator = op;
  line->next = NULL;
  line->variable2 = NULL;
  line->isNext = 0;

  addToQueue(line);
}

void createInstruction(char* variable,int operand1,int isVar1, int operand2,
int isVar2, int operator){

  createInstruction0(variable,operand1,isVar1,operand2,isVar2,operator,0);
}
void createInstruction0(char* variable,int operand1,int isVar1, int operand2,
int isVar2, int operator,
    int thereIsElse){


  TacLine* line = (TacLine*)malloc(sizeof(TacLine));
  line->variable = variable;
  line->variable2 = NULL;
  line->operand1 = operand1;
  line->isVar1 = isVar1;
  line->operand2 = operand2;
  line->isVar2 = isVar2;
  line->operator = operator;
  line->isRegisterFunctionCall = 0;
  line->isSimple = 0;
  line->isStatement = 0;
  line->isVariableEq = 0;
  line->thereIsElse = thereIsElse;
  line->next = NULL;
  line->isNext = 0;

  if(!isVar1){

    int tempIsVar = line->isVar1;
    int tempOperand = line->operand1;

    line->isVar1 = line->isVar2;
    line->operand1 = line->operand2;

    line->isVar2 = tempIsVar;
    line->operand2 = tempOperand;
  }

  addToQueue(line);
}

void printTabs(int tabs){

  for(int t = 0; t < tabs; t++)printf(”\t”);
}

void printSimpleAssignment(char* var, int value, int isVar){

  printf(”%s␣=␣”,var);

  if(isVar){
    printf("$t%d",value);

  }else{

    printf("%d",value);
  }

  printf(";\n");
}

void printTacLine(char* var,int op, struct TypeValue left ,
struct TypeValue right){

  //printf("t%d = ",var);
  printf("%s␣=␣",var);

  printOperand(left ,0);
  if(op == EQ_OP){

    printf("␣==␣");
  }else{

    printf("␣%c␣",op);
  }

  printOperand(right ,1);

  printf(";\n");
}

void printOperand(struct TypeValue operand,int isRight){

  if(operand.type == 0){

    printf("%d",operand.value);

  }else if(operand.type == 1){

    printf("$t%d",operand.value);

  }else if(operand.type == 3){

    printf("%s",operand.lexeme);
  }
}

void resetTemp(){

  temp_count = 0;
}

int genTemp(){

  return ++temp_count;
}

int genLabelCount(){

  return ++label_count;
```

```
}

int getLabelCount(){

  return label_count;
}

//for nested if and while statements
//declares the end of a if and while statement
//if_1
//if_2
//..code..
//if_2
//releaseLabelCount()
//if_1
void releaseLabelCount(){

  --label_count;
}

int reuseTemp(){

  return temp_count;
}


int strNumSize(int value,int strLength){

  int digitsNo = numDigits(value);
  return strLength * value * sizeof(char);

}

//merges two strings and an integer
void intToString(char* start,int value,char* end,char** combine){

  int digitsNo = numDigits(value);
  char num[digitsNo];
  itoa(value,num,10);

  *combine = (char*)malloc(sizeof(char)*(digitsNo
  + strlen(start) + strlen(end) + 1));
  strcpy(*combine,start);
  strcat(*combine,num);
  strcat(*combine,end);
}


int numDigits(int number)
{
    int digits = 0;
    if (number < 0)
    {
      digits = 1;
    }
    while (number) {
        number = number / 10;
        digits++;
    }
    return digits;
```

```
}


void compile(NODE* tree){

  compile0(tree,0);

}

struct FunctionNameNode{

  char* functionName;
  struct FunctionNameNode* next;
  struct FunctionNameNode* last;
};

typedef struct FunctionNameNode FunctionNameNode;

struct AssemblyContext{

  int paramNo;
  int isParam;
  //char* currentFunction;
  FunctionNameNode* head;
  FunctionNameNode* current;
};

typedef struct AssemblyContext AssemblyContext;

void pushFunctionName(char* functionName,AssemblyContext* context);
void popFunctionName(AssemblyContext* context);
char* getBlockNo(char* totalName);
void printNewBlock(char* blockName);
void printEndBlock(char* blockName);
void convertToAssembly(TacLine* line,AssemblyContext* context);
void printFindClosure(int closureNo);
void printLw(char* variable, char* variable2,int loadAddress);
void printConditionStatement(TacLine* line);
void printReturnStatementInstruct(void* retValue, int isVar,
int isTemp,AssemblyContext* context);
void printPopArgInstruct(char* variable,AssemblyContext* context);
void printFunctionCall(char* function,int temp,AssemblyContext* context,
int callingTemp);
char* getFunctionNameFromEndTag(char* variable);
void printParamInstruct(void* param,int type,AssemblyContext* context);
void printAssemInstruct(char* instruct,TacLine* line,AssemblyContext* context);
void printAssemOperand(void* operand,int isVar,int isStr);
void createParamData(int numOfParams);
void compileToAssembly(NODE* tree,int optimize);


void pushFunctionName(char* functionName,AssemblyContext* context){

  if(context->head == NULL){

    context->head = (FunctionNameNode*)malloc(sizeof(FunctionNameNode));
    context->head->functionName = functionName;
    context->head->last = NULL;
    context->head->next = NULL;
    context->current = context->head;
  }else{

    FunctionNameNode* next = context->head;

    while(next->next != NULL){

      next = next->next;
    }

    next->next = (FunctionNameNode*)malloc(sizeof(FunctionNameNode));
    next->next->functionName = functionName;
    next->next->last = next;
    next->next->next = NULL;
    context->current = next->next;
  }
}

void popFunctionName(AssemblyContext* context){

  if(context->current != NULL){

    FunctionNameNode* cleanUp = context->current;

    if(cleanUp->last == NULL){

      context->head = NULL;
      context->current = NULL;

    }else{

      context->current = cleanUp->last;
    }

    free(cleanUp);
  }
}

//gets the block no from a if or while name
char* getBlockNo(char* totalName){

  int i = strlen(totalName);
  int start;
  int end = i;

  while(i >=0){

    if(totalName[i] == '_'){

      start = i+1;
      break;
    }

    i--;
  }

  char* blockNo = (char*)malloc(sizeof(char) * (end - start));

  for(i = start; i <= end; i++){
    blockNo[i-start] = totalName[i];
  }

  return blockNo;

}

//prints a new while or if block
void printNewBlock(char* blockName){

  char* blockNo = getBlockNo(blockName);
  int closureNo;
  int offset = addNextMemLoc(blockNo,1,&closureNo).valueType.intValue;
  pushStack(getEnvironment(),"");
  printf("sw␣$fp,␣%d($fp)\n",offset);
  printf("add␣$fp,␣$fp,␣%d\n",offset);
  addNextMemLoc("stub",1,&closureNo);
}

//prints a end while or if block
void printEndBlock(char* blockName){

  int closureNo;
  int offset = getValueByEquality(getBlockNo(blockName),
  &closureNo).valueType.intValue;

  printf("lw␣$fp,␣0($fp)\n",offset);
  popStack();
}


//main function that converts to assembly
void convertToAssembly(TacLine* line,AssemblyContext* context){

  char* instruct;
  int hasPrinted = 0;

  if(line->operator != 'A' && !context->isParam){

    context->paramNo = 0;
  }

  switch(line->operator){

  case '+':
    instruct = "add";
    break;
  case '-':
    instruct = "sub";
    break;
  case '*':
    instruct = "mul";
    break;
  case '/':
    instruct = "div";
    break;
  case '<':
    instruct = "slt";
    break;
  case '>':
```

```
  instruct = "sgt";
  break;
case EQ_OP:
  instruct = "seq";
  break;
case '=':
  if(line->isVariableEq){

    if(line->isVar1Temp){

      instruct = "lw";

    }else{

      instruct = "sw";
    }

  }else{

    if(line->isVar1){

      instruct = "move";

    }else{

      instruct = "li";
    }

  }
  break;
//if statement
case 'C':

  printConditionStatement(line);
  if(line->thereIsElse){
    printNewBlock(line->variable);
  }
  hasPrinted = 1;
  break;
//while statement
case 'W':

  printConditionStatement(line);
  printNewBlock(line->variable);
  hasPrinted = 1;

  break;

//parameter
case 'P':

  if(line->paramType == 3){

    printParamInstruct(line->variable, line->paramType, context);

  }else{

    printParamInstruct(&line->operand1, line->paramType, context);

  }
```

```
      hasPrinted = 1;
      context->isParam = 1;
      context->paramNo++;
      break;

    //function call
    case 'F':
      if(line->isRegisterFunctionCall){

        printFunctionCall(line->variable,line->operand1,context,1);

      }else{

        printFunctionCall(line->variable,line->operand1,context,0);
      }

      hasPrinted = 1;
      context->paramNo = 0;
      break;
    //argument being passed
    case 'A':
      printPopArgInstruct(line->variable,context);
      hasPrinted = 1;
      context->isParam = 0;
      context->paramNo++;
      break;
    //return statement
    case 'R':

      if(line->isVar1){

        printReturnStatementInstruct(line->variable,line->isVar1,
    line->isVar1Temp,context);

      }else{

        printReturnStatementInstruct(&line->operand1,line->isVar1,
        line->isVar1Temp,context);
      }

      hasPrinted = 1;
      break;
    }

    //some cases print there own cases, some use the generic printAssemInstruct
    if(!hasPrinted){
      printAssemInstruct(instruct,line,context);
    }

  }

  //prints a search for a variable in a closure
  //$v1 will contain the frame pointer for the closure
  //closureNo of places back
  void printFindClosure(int closureNo){

    if(closureNo > 0){

      int closureNo;
      printf("move␣$v1,␣$fp\n");
    }
    int i = closureNo;
    while(i-- > 0){

      printf("lw $v1, 0($v1)\n");
    }

  }

  //generic print a load word
  void printLw(char* variable, char* variable2,int loadAddress){

    int closureNo;
    Value offset = getValueByEquality(variable2,&closureNo);

    //get integer
    if(offset.isFunction == 0){

      printFindClosure(closureNo);

      if(closureNo > 0){

        printf("lw %s %d($v1)",variable,offset.valueType.intValue);

      }else{

        printf("lw %s %d($fp)",variable,offset.valueType.intValue);
      }

    }else{

      //creating a function pointer
      printf("li $v0, 9\nli $a0, 8\nsyscall\n");
      printf("la $t0, %s\n",variable2);
      printf("sw $t0, ($v0)\n");
      printf("sw $fp, 4($v0)\n");
      printf("la %s, ($v0)",variable);

    }
  }



  void printConditionStatement(TacLine* line){

    printf("beq ");
    printAssemOperand(&line->operand1, line->isVar1,0);
    printAssemOperand(&line->operand2, line->isVar2,0);
    printf("%s\n",line->variable);
  }

  void printReturnStatementInstruct(void* retValue, int isVar,
    int isTemp,AssemblyContext* context){

    if(isVar){

      printLw("$a0",(char*)retValue,1);
      printf("\n");

    }else if(isTemp){
      printf("move␣$a0,␣$t%d\n",*((int*)retValue));

    }else{

      printf("li␣$a0,␣%d\n",*((int*)retValue));
    }

    //get $ra and $fp
    //$fp first so that we keep track of the function pointer before getting $fp
    printLw("$ra","$ra",0);
    printf("\n");
    printLw("$fp","$fp",0);
    printf("\n");

    printf("jr␣$ra\n");

  }

  void printPopArgInstruct(char* variable,AssemblyContext* context){

    printf("lw␣$t0,␣%d($a1)\n",context->paramNo*4);
    printf("sw␣$t0,%d($fp)\n",(context->paramNo*4) + 8);
    int closureNo;
    addNextMemLoc(variable,1,&closureNo);
  }

  //calling a function pointer
  void printCallFunctionPointer(char* function){

    int closureNo;
    Value offset = addNextMemLoc("$s2",1,&closureNo);
    printf("sw␣$s2,%d($fp)\n",offset.valueType.intValue);
    printf("move␣$s2,␣$fp\n");
    printf("lw␣$fp,␣4(%s)\n",function);
    printf("lw␣$t0,␣0(%s)\n",function);
    printf("jalr␣$t0\n");
    printf("move␣$fp,␣$s2\n");
    printf("lw␣$s2,␣%d($fp)\n",offset.valueType.intValue);


  }

  void printFunctionCall(char* function,int temp,AssemblyContext* context,
  int callingTemp){

    char* parent = getParent(function);
    if(callingTemp){

      printCallFunctionPointer(function);

    }else if(isGlobalFunction(function)){

      printf("jal␣%s\n",function);

    }else{

      int closureNo;
      Value offset = getValueByEquality(function,&closureNo);
      //calling child function from parent function
      if(offset.isFunction == 1){
        if(offset.valueType.intValue < 0){

          printf("jal %s\n",function);
        }else{

          //function is parent function
          printFindClosure(closureNo);
          printf("lw $v1, %d($v1)\n",offset.valueType.intValue);
          printf("jalr $v1\n");
        }
      }else{

        //calling function pointer
        if(closureNo > 0){
          printFindClosure(closureNo);
          printf("lw $v1, %d($v1)\n",offset.valueType.intValue);
        }else{

          printf("lw $v1, %d($fp)\n",offset.valueType.intValue);
        }

        printCallFunctionPointer("$v1");
      }


    }

    //get the returned value
    printf("move $t%d, $a0\n",temp);
  }

  //get the function name from the end tag of a function
  char* getFunctionNameFromEndTag(char* variable){

    int funcNameSize = (strlen(variable) - strlen(".end "))+1;
    char* functionName = (char*)malloc(funcNameSize * sizeof(char));

    int f = 0;
    for(int f = 0; f < funcNameSize-1; f++){

      functionName[f] = variable[f+strlen(".end ")];
    }

    functionName[funcNameSize-1] = '\0';

    return functionName;
  }

  void printParamInstruct(void* param,int type,AssemblyContext* context){

    if(context->paramNo == 0){
      printf("la $a1, params\n");
    }

    printf("sw $t%d, %d($a1)\n",*((int*)param),context->paramNo*4);

  }

  void printAssemInstruct(char* instruct,TacLine* line,AssemblyContext* context){
```

```
if(line->isStatement){

  //new function
  if(line->isStatement == 2){

    //place function name on the current frame
    addFunctonToFrame(line->variable);
    //if this function has a parent then create a jump statement
    if(hasParent(line->variable)){

      printf("j␣end%s\n",line->variable);

      pushStack(getEnvironment(),"");

    }else{

      //push a new stack for each function
      //main has global frame
      if(strcmp(line->variable,MAINSTRING) != 0){

        pushStack(NULL,"");
      }

    }

    pushFunctionName(line->variable,context);
    printf("%s:\n",line->variable);

    int bytesToAll = getBytesToAllocation(line->variable);

    //allocate function frame
    //place old frame pointer, returned address in frame
    printf("li␣$v0,␣9\n");
    printf("li␣$a0,%d\n",bytesToAll);
    printf("syscall\n");
    printf("sw␣$fp,␣($v0)\n");
    printf("sw␣$ra,␣4($v0)\n");
    int closureNo;
    //remember where the $fp and $ra are
    addNextMemLoc("$fp",1,&closureNo);
    addNextMemLoc("$ra",1,&closureNo);

    printf("move␣$fp,␣$v0");

  //jump statements for while and if statement
  }else{

    //except for this which is the end of function
    if(line->isStatement >= 3){

      printf("%s",line->variable);

      if(line->isStatement == 4){

        popFunctionName(context);
        char* functionName = getFunctionNameFromEndTag(
                  line->variable);
        int needJump = hasParent(functionName);
```

```
          if(needJump){

            printf("\nend%s:",functionName);
          }
        }

        popStack();

      }else{

        if(line->operator == 'J'){

          printEndBlock(line->variable);
          printf("%s:",line->variable);

        }else if(line->operator == 'K'){

          printf("%s:\n",line->variable);
          printNewBlock(line->variable);

        }else if(line->operator == 'M'){

          printEndBlock(line->variable);
          printf("%s",line->variable);

        }else if(line->operator == 'O'){

          printf("%s",line->variable);

        }else{

          printf("%s:",line->variable);
        }
      }

    }

  }else if(line->isVariableEq){

    //store a value in a register
    if(strcmp(instruct,"sw") == 0){

      int closureNo = 0;
      int value = addNextMemLoc(line->variable,
      line->isVariableCreation,&closureNo).valueType.intValue;
      printFindClosure(closureNo);

      if(closureNo > 0){

        printf("%s␣%s␣%d($v1)",instruct,line->variable2,value);

      }else{

        printf("%s␣%s␣%d($fp)",instruct,line->variable2,value);
      }


    //load a value into a register
    }else if(strcmp(instruct,"lw") == 0){
        printLw(line->variable,line->variable2,0);
      }

  }else{
    printf("%s␣",instruct);
    if(line->variable != IFSTRING){
      printAssemOperand(line->variable,1,1);
    }
    printAssemOperand(&(line->operand1),line->isVar1,0);
    if(!line->isSimple){

      printAssemOperand(&(line->operand2),line->isVar2,0);
    }
  }
  printf("\n");

}

//structure that represents a register name that needs to be replaced
struct ReplaceWith{

  int toReplace;
  int replaceWith;
  struct ReplaceWith* next;
};

typedef struct ReplaceWith ReplaceWith;

//check if any of the element in a line need to be replaced
//offset is which element in the list to compare the line to
//the list is the list of registers to be replaced
int* toReplace(ReplaceWith** list, TacLine* line,int offset){

  ReplaceWith* next = (*list);

  //select the element in the list to compare the line to
  while(offset > 0 && next != NULL){

    next = next->next;
    offset--;
  }

  //if there are not that many elements in the list then return null
  if(next == NULL){

    return NULL;
  }

  //create a array of 4 elements
  int* needToReplace = (int*)malloc(sizeof(int)*4);

  while(next != NULL){

    //place the register to replace with in the last element of the array
    needToReplace[3] = next->replaceWith;
    //if the assigned value is register to replace then indicate that it needs replacing
    if(line->variable != NULL &&
    line->variable[2] - '0' == next->toReplace){

      needToReplace[0] = 1;
    }else{

      needToReplace[0] = 0;
    }

    //if the first operand is a register that needs to be
    //replaced then indicate that it needs replacing
    if(line->operand1 == next->toReplace ){

      needToReplace[1] = 1;

    }else{

      needToReplace[1] = 0;
    }
    //if the second operand is a register that needs to
    //be replaced then indicate that it needs replacing
    if(line->operand2 == next->toReplace){

      needToReplace[2] = 1;

    }else{

      needToReplace[2] = 0;
    }

    //return the array if something has to be replaced
    if(needToReplace[0] || needToReplace[1] || needToReplace[2]){

      return needToReplace;

    }else{

      //otherwise keep looking though the list
      next = next->next;
    }

  }

  //if nothing needs replacing then return null
  free(needToReplace);
  return NULL;
}

void optimizeTacCode(TacLine* first){

  //create the list of registers that need to be replace
  ReplaceWith** list = (ReplaceWith**)malloc(sizeof(ReplaceWith*));
  (*list) = NULL;
  TacLine* next = first;

  //go though the entire TAC code
  while(next != NULL){

    //if there is a statement then the compiler is in a new basic block
    if(next->isStatement > 0){

      //clear list
      (*list) = NULL;//small memory leak
```

```c
  next = next->next;
  continue;
}

//if this TAC line is a $tn = $tm statement
if(next->isSimple == 1 && next->isStatement == 0 &&
    (next->variable != NULL && next->variable[0] == '$')
    && next->isVar1 == 1){

  //set this TAC line so that it is not executed by the MIPS compiler
  next->deleteInOptimization = 1;
  //if the list is empty then add the ReplaceWith structure to the head
  if((*list) == NULL){

    //set the left hand register as the register to be replaced
    //set the right hand register as the register to replace it with
    (*list) = (ReplaceWith*)malloc(sizeof(ReplaceWith));
    (*list)->toReplace = next->variable[2] - '0';
    (*list)->replaceWith = next->operand1;
    (*list)->next = NULL;

  //else add the ReplaceWith structure add the end of the list
  }else{

    ReplaceWith* append = (*list);

    while(append->next != NULL){

      append = append->next;
    }

    append->next = (ReplaceWith*)malloc(sizeof(ReplaceWith));
    append->next->toReplace = next->variable[2] - '0';
    append->next->replaceWith = next->operand1;
    append->next->next = NULL;
  }

}else{

  //else compile line when compiling MIPS
  next->deleteInOptimization = 0;
}

int n = 0;
//look though the replacement list
while(1){

  //get array of replacement information
  //n is the replacement elements that needs to be looked at next
  //if n wasn't incremented then the toReplace would look at
  //the same element each time
  int* replacement = toReplace(list,next,n);
  n++;
  if(replacement == NULL){

    //if there is nothing to replace
    break;

  }else{
        //replace register
        if(replacement[0]){

          next->variable[2] = replacement[3] + '0';
        }

        if(replacement[1]){

          next->operand1 = replacement[3];
        }

        if(replacement[2]){

          next->operand2 = replacement[3];
        }
      }
    }

    //move on to next tacline
    next = next->next;
  }
}



void printAssemOperand(void* operand,int isVar,int isStr){

  if(isVar && isStr){

    printf("%s␣",operand);

  }else if(isVar){

    printf("$t%d␣",*((int*)operand));

  }else{

    printf("%d␣",*((int*)operand));
  }
}



void createParamData(int numOfParams){

  if(numOfParams > 0){
    printf(".data\nparams:␣␣.word␣");
    int i;
    for(i = 0; i < numOfParams-1; i++)printf("0,␣");
    printf("0\n");
  }

}

void compileToAssembly(NODE* tree,int optimize){

  compile0(tree,0);


  printf("\n\n\n");
  //calculate function meta-data
  calculateFunctionInfo(getElement(0));
  createParamData(getMaxParams());

  printf(".text\n.globl\tmain\n");
  printf("_main:\njal_main\nli_$v0,10\nsyscall\n.end__main\n");
  //optimize code
  AssemblyContext* context = (AssemblyContext*)malloc(sizeof(AssemblyContext));
  if(optimize){
    optimizeTacCode(getElement(0));
  }
  context->head = NULL;
  context->paramNo = 0;

  //create global frame
  pushStack(NULL,"");
  int i;
  //turn tac lines into mips code
  for(i = 0; i < getSize(); i++){

    TacLine* line = getElement(i);
    if(line->deleteInOptimization != 1){
      convertToAssembly(getElement(i),context);
    }
  }


}
```

2 compile.h

```
/*
 * compile.h
 *
 *  Created on: 12 Oct 2016
 *      Author: harry
 */
#include "interpret.h"

#ifndef COMPILE_H_
#define COMPILE_H_



#endif /* COMPILE_H_ */
```

3 frames.c

```
/*
 * frames.c
 *
 *  Created on: 12 Oct 2016
 *      Author: harry
 */

#include <stdlib.h>
#include <stdio.h>
#include "nodes.h"

struct Frame{
  struct SymbolNode* listHead;
  struct Frame* next;
  struct Frame* last;
  struct Frame* closure;
  char* functionName;
  int no;
};

typedef struct Frame Frame;

Frame* currentFrame;
Frame* globalFrame;


struct Closure{

  NODE* functionBody;
  char* parentFunctionName;
  Frame* env;
};

union ValueType{

  int intValue;
  struct Closure* closure;
};

struct Value{

  int isFunction;
  union ValueType valueType;
};

typedef struct Value Value;

struct SymbolNode{

  char* symbol;
  Value value;
  struct SymbolNode* next;
  int closureNo;
};

typedef struct Value Value;
typedef union ValueType ValueType;

void getValueFromFrame(Frame* frame,char* symbol,int comparePointer,
    struct SymbolNode** finalResult);
void addSymbol0(char* symbol,Value value,int comparePointer,int isVariableCreation);
void printFrame(Frame* frame);
void pushStack(Frame* env,char* functionName);
void popStack();
Frame* getEnvironment();
void addSymbol(char* symbol, Value value,int isVariableCreation);
void addSymbolByEquality(char* symbol, Value value,int isVariableCreation);
int containsSymbol(char* symbol);
int isOnGlobalFrame(char* symbol);
struct SymbolNode* getValue0(char* symbol,int comparePointer);
void getValueFromFrame(Frame* frame,char* symbol,int comparePointer,
    struct SymbolNode** finalResult);
Value getLastValue();
void changeAllInFrame(int amount);
Value getValueByEquality(char* symbol,int* closureNo);
Value getValue(char* symbol);


//prints a frame used for debugging
void printFrame(Frame* frame){

  if(frame == NULL){

    printf("frame␣is␣null\n");

  }else{

    printf("Print␣Bindings␣begin␣\n");
    struct SymbolNode* tranverse = frame->listHead;
    while(tranverse != NULL){

      printf("\tBinding␣%s␣",tranverse->symbol);
      printf("%d:\n",tranverse->value.valueType.intValue);
      tranverse = tranverse->next;
    }
    printf("Print␣Bindings␣end\n");
  }
}

//creates a new frame with this frame being connected to the last one
//frame* evn is the parent function frame if this is a child function
//if this is a normal function then env will be null
void pushStack(Frame* env,char* functionName){


  if(currentFrame == NULL){

    currentFrame = (Frame*)malloc(sizeof(Frame));
    currentFrame->listHead = NULL;
    currentFrame->last = NULL;
    currentFrame->no = 1;
    currentFrame->next = NULL;
    currentFrame->functionName = functionName;
    currentFrame->closure = NULL;
    globalFrame = currentFrame;

  }else{

    Frame* nextFrame = (Frame*)malloc(sizeof(Frame));
    nextFrame->last = currentFrame;
    nextFrame->listHead = NULL;
    nextFrame->no = currentFrame->no + 1;
    nextFrame->closure = env;
    currentFrame->next = nextFrame;
    currentFrame->functionName = functionName;
    currentFrame = nextFrame;
  }

}

//goes back to the last frame
```

```
void popStack(){
  Frame* lastFrame = currentFrame;
  currentFrame = currentFrame->last;

}

//gets the current environment
Frame* getEnvironment(){

  return currentFrame;
}

//adds a symbols to the current frame using pointer comparision
void addSymbol(char* symbol, Value value,int isVariableCreation){

  addSymbol0(symbol,value, 1,isVariableCreation);
}

//adds a symbols to the current frame using equality of strings
void addSymbolByEquality(char* symbol, Value value,int isVariableCreation){

  addSymbol0(symbol,value,0,isVariableCreation);
}

//if the current frame contains a symbol
int containsSymbol(char* symbol){

  if(currentFrame == NULL || currentFrame->listHead == NULL){

    return 0;
  }

  struct SymbolNode* tranverse = currentFrame->listHead;

  while(tranverse != NULL){

    if(strcmp(tranverse->symbol,symbol) == 0){

      return 1;
    }

    tranverse = tranverse->next;
  }

  return 0;

}

//adds a symbol to the current mapping it to the value in value
void addSymbol0(char* symbol,Value value,int comparePointer,int isVariableCreation){

  //if the symbol already exists
  struct SymbolNode* update = getValue0(symbol,comparePointer);

  //if the symbol doesn't already exist or this symbol is to be added on to this scope anyway
  if(update == NULL || isVariableCreation){

    //add the symbol
    if(currentFrame->listHead == NULL){
      currentFrame->listHead = (struct SymbolNode*)malloc(sizeof(struct SymbolNode));
      currentFrame->listHead->symbol = symbol;
      currentFrame->listHead->value = value;
      currentFrame->listHead->next = NULL;

    }else{

      struct SymbolNode* tranverse = currentFrame->listHead;
      while(tranverse->next != NULL){

        tranverse = tranverse->next;

      }

      tranverse->next = (struct SymbolNode*)malloc(sizeof(struct SymbolNode));
      tranverse->next->symbol = symbol;
      tranverse->next->value = value;
      tranverse->next->next = NULL;

    }
  }else{

    //update the value with this symbol
    update->value = value;
  }

}


//if a symbol is on the global frame and is a function
int isOnGlobalFrame(char* symbol){

  struct SymbolNode* next = globalFrame->listHead;

  while(next != NULL){

    if(strcmp(next->symbol,symbol) == 0 && next->value.isFunction == 1){

      return 1;
    }

    next = next->next;
  }

  return 0;
}


//get a value from the current frame or from a parent's frame(recursively)
struct SymbolNode* getValue0(char* symbol,int comparePointer){

  Frame* frame = currentFrame;
  struct SymbolNode* finalResult = NULL;

  //try this frame
  getValueFromFrame(frame,symbol,comparePointer,&finalResult);

  Frame* thisClosureEnv = frame->closure;
  int closureNo = 0;
  //if not this frame then try the parent's frame, then it's parent's frame etc...
  while(finalResult == NULL && thisClosureEnv != NULL){

    getValueFromFrame(thisClosureEnv,symbol,comparePointer,&finalResult);
    thisClosureEnv = thisClosureEnv->closure;
    closureNo++;
  }

  //check the global frame
  if(finalResult == NULL){

    getValueFromFrame(globalFrame,symbol,comparePointer,&finalResult);
  }

  //keep track of which closure this symbol was in
  if(finalResult != NULL){
    finalResult->closureNo = closureNo;
  }
  return finalResult;
}

//trys to find a value in a frame
void getValueFromFrame(Frame* frame,char* symbol,int comparePointer,
    struct SymbolNode** finalResult){

  struct SymbolNode* tranverse = frame->listHead;
  if(tranverse != NULL){

    //either compare by pointer or by equality
    //stop if there are no more mappings to search or the correct mapping
    //has been found
    while((tranverse->next != NULL && ((tranverse->symbol != symbol)
        || (!comparePointer && strcmp(tranverse->symbol,symbol) != 0)))){

      tranverse = tranverse->next;
    }

    //if the correct mapping was found
    if((comparePointer && tranverse->symbol == symbol)
        || (!comparePointer && strcmp(tranverse->symbol,symbol) == 0)){

      *finalResult = tranverse;
    }
  }
}

//get the last value added to the current frame
Value getLastValue(){

  struct SymbolNode* node = currentFrame->listHead;

  if(node == NULL){

    Value value;
    value.valueType.intValue = -4;
    return value;
  }

  while(node->next != NULL){
    node = node->next;
  }

  return node->value;
}

//changes all of the values in the current frame by a given amount
void changeAllInFrame(int amount){

  struct SymbolNode* node = currentFrame->listHead;

  while(node != NULL){

    node->value.valueType.intValue += amount;
    node = node->next;
  }
}

//get a value by equality of the symbol
Value getValueByEquality(char* symbol,int* closureNo){

  struct SymbolNode* sym = getValue0(symbol,0);
  *closureNo = sym->closureNo;
  return sym->value;
}

//get a value by pointer comparision of the symbol
Value getValue(char* symbol){

  return getValue0(symbol,1)->value;
}
```

4 frames.h

```
/*
 * frame.h
 *
 *  Created on: 12 Oct 2016
 *      Author: harry
 */

#include "nodes.h"
#include "C.tab.h"

struct Closure{

  NODE* functionBody;
  char* parentFunctionName;
  struct Frame* env;
};
union ValueType{

  int intValue;
  struct Closure* closure;

};

struct Value{

  int isFunction;
  union ValueType valueType;
};

typedef struct Value Value;

struct SymbolNode{

  char* symbol;
  Value value;
  struct SymbolNode* next;
  int closureNo;
};

struct Frame{

  struct SymbolNode* listHead;
  struct Frame* next;
  struct Frame* last;
  struct Frame* closure;
  char* functionName;
  int no;
};

typedef struct Frame Frame;

typedef union ValueType ValueType;
typedef struct Closure Closure;

void pushStack(struct Frame* env,char* functionName);
void popStack();
void addSymbol(char* symbol,Value value,int isVariableCreation);
void addSymbolByEquality(char* symbol, Value value,int isVariableCreation);
Frame* getEnvironment();
//void addSymbol0(char* symbol,Value value,int isClosure);
Value getValue(char* symbol);
void changeAllInFrame(int amount);
Value getValueByEquality(char* symbol,int* closureNo);
int containsSymbol(char* symbol);
Value getLastValue();
Value isFunctionPointer(char* symbol);
```

5 interpret.c

```
/*
 * interpret.c
 *
 *  Created on: 6 Oct 2016
 *      Author: harry
 */

#include <stdio.h>
#include <ctype.h>
//#include "nodes.h"
#include "C.tab.h"
#include <string.h>
#include "frames.h"
#include <stdlib.h>

//represents a parameter in a parameter list
struct Parameter{
  char* symbol;
  Value value;
  struct Parameter *last;
  struct Parameter *next;
};

typedef struct Parameter Parameter;

void parseParameters(NODE* parameter,NODE* argument,Parameter** nextParam);
int interpret(NODE* tree);
Value interpret0(NODE* tree,int* answerBranch);
Value interpret1(NODE* tree,int* answerBranch,int variableCreated);
Value evalFunction(NODE* tree);
Value evalExp(NODE* tree);
int evalCondition(NODE* tree);

//start point
int interpret(NODE* tree){

  printf("\n\n");
  int* answerBranch = (int*)malloc(sizeof(int));

  //creates global frame where function definitions are stored
  pushStack(NULL,"main");
  Value value = interpret1(tree,answerBranch,0);
  free(answerBranch);
  //destroy global frame
  popStack();

  return value.valueType.intValue;
}

Value interpret0(NODE* tree,int* answerBranch){

  return interpret1(tree,answerBranch,0);
}

Value interpret1(NODE* tree,int* answerBranch,int variableCreated){

  Value found;

  //if the node is a function
  if(tree->type == 'D' ){

    TOKEN* function = ((TOKEN*)tree->left->right->left->left);

    //if it's not the main the store the environment and the code
    if(strcmp(function->lexeme,"main") != 0){

      Value functionVal;
      Closure* closure = (struct Closure*)malloc(sizeof(struct Closure));
      //create closure by taking the current environment(frame)
      closure->env = getEnvironment();

      closure->functionBody = tree;
      functionVal.valueType.closure = closure;
      functionVal.isFunction = 1;
      //add the function name to the frame so it can be recalled later
      addSymbol(function->lexeme,functionVal,1);
      *answerBranch = 0;
      Value ret;
      ret.isFunction = 0;
      ret.valueType.intValue = 0;
      return ret;

    }

  }

  if(tree->type == RETURN){

    //return the value of the return statement and set this branch to an answer branch
    *answerBranch = 1;
    Value value = evalExp(tree);
    return value;


  }else if(tree->type == '='){

    Value evalValue = evalExp(tree->right);

    *answerBranch = 0;
    //do an assignment
    //if a ~ was a parent of this node then this is a variable creation
    //variable creations are pushed on to this frame regardless if there is a variable
    //of the same name on the frame
    addSymbol(((TOKEN*)tree->left->left)->lexeme,evalValue,variableCreated);

    return evalValue;

  }else if(tree->type == IF){

    //create closure around the if statement
    pushStack(getEnvironment(),"");
    if(evalCondition(tree->left)){

      //if code is executed
      Value ret;
      //if there is no else code
      if(tree->right->type == ELSE){

        ret = interpret0(tree->right->left ,answerBranch);

      }else{

        ret = interpret0(tree->right,answerBranch);
      }

      popStack();
      return ret;

    }else if(tree->right->type == ELSE){

      //execute else code
      Value ret = interpret0(tree->right->right,answerBranch);
      popStack();
      return ret;

    }else{
      //there is no else code and the if code was no executed
      *answerBranch = 0;
      Value value;
      value.isFunction = 0;
      value.valueType.intValue = 0;
      popStack();
      return value;
    }


  }else if(tree->type == WHILE){

    NODE* condition = tree->left;
    NODE* loopCode = tree->right;
    //place closure around while loop
    pushStack(getEnvironment(),"");
    while(evalCondition(condition)){

      //interpret the while loop's code in till the condition is false
      Value possRet = interpret0(loopCode,answerBranch);

      //if there is a return statement inside the while statement
      if(*answerBranch == 1){

        return possRet;
      }
    }

    popStack();
    //there was no answer branch inside the while loop
    *answerBranch = 0;

    Value zero;
    zero.isFunction = 0;
    zero.valueType.intValue = 0;

    return zero;

  //if we need to keep tree walking
  }else if(tree->type != LEAF){

    *answerBranch = 0;
    int* leftBranch = (int*)malloc(sizeof(int));
    *leftBranch = 0;

    //check if there is an answer in the left branch
    if(tree->left != NULL){

      Value leftAnswer = interpret1(tree->left ,leftBranch ,tree->type == '~');

      if(*leftBranch){

        found = leftAnswer;
        *answerBranch = 1;
        free(leftBranch);

      }else{

        *answerBranch = 0;
      }
    }

    //if there wasn't an answer in the left branch then try the right branch
    if(tree->right != NULL && (*leftBranch) == 0){

      int* rightBranch = (int*)malloc(sizeof(int));
      Value rightAnswer = interpret1(tree->right,rightBranch,tree->type == '~');

      if(*rightBranch){

        found = rightAnswer;
        *answerBranch = 1;
        free(rightBranch);

      }else{

        *answerBranch = 0;
      }
    }

  }

  //return the answer in either the right or left branch if any
  return found;

}


//evaluate a function call
Value evalFunction(NODE* tree){

  Value functionVal;
  char* functionName = ((TOKEN*)tree->left->left)->lexeme;
  //if this is a function calling the results of another functions then call that function
  if(tree->left->type == APPLY){

    functionVal = evalFunction(tree->left);

  }else{

    //else get the function from the frame/closure
    functionVal = getValue(functionName);

  }

  Closure* function = functionVal.valueType.closure;

  //create list of parameters
  Parameter** paramList = (Parameter**)malloc(sizeof(Parameter*));
  *paramList = NULL;
  NODE* functionBody = function->functionBody;

  //find the parameters in the tree
  if(functionBody->left->right->right != NULL){
    parseParameters(functionBody->left->right->right,tree->right,paramList);
  }
  //create a new frame for this function
  pushStack(function->env,((TOKEN*)tree->left->left->left)->lexeme);
  //add the arguments to that frame
  while((*paramList) != NULL){

    addSymbol((*paramList)->symbol,(*paramList)->value,1);
    *paramList = (*paramList)->last;
  }

  //evaluate the function and return the result, as well as the answer branch
  int* answerBranch = (int*)malloc(sizeof(int));
  Value retValue = interpret0(functionBody->right,answerBranch);
  popStack();
  return retValue;

}


//collect the called function's parameters
void parseParameters(NODE* parameter,NODE* argument,Parameter** paramList){

  //keep looking for more parameters
  if(parameter->type == ','){

    parseParameters(parameter->left,argument->left,paramList);
    parseParameters(parameter->right,argument->right,paramList);

  }else{

    //add the parameter to the parameters list
    TOKEN* parToken = (TOKEN*)parameter->right->left;
    //print_tree(parameter);
    Value value = evalExp(argument);
    if(*paramList == NULL){

      *paramList = (Parameter*)malloc(sizeof(Parameter));
      (*paramList)->last = NULL;

    }else{

      //keeps a double linked list
      (*paramList)->next = (Parameter*)malloc(sizeof(Parameter));
      (*paramList)->next->last = *paramList;
      *paramList = (*paramList)->next;

    }

    (*paramList)->symbol = parToken->lexeme;
    (*paramList)->value = value;

  }
}

//evaluates an arithmetic expression
Value evalExp(NODE* tree){

  Value zero;
  zero.isFunction = 0;
  zero.valueType.intValue = 0;
  //if there is not assignment return zero
  if(tree == NULL) return zero;
  //if there is a function call, call it
  if(tree->type == APPLY) return evalFunction(tree);
  //if this is a leaf
  if(tree->type == LEAF){

    //either get the value of a variable
    TOKEN* leaf = (TOKEN*)tree->left;
    if(leaf->type == IDENTIFIER){

      Value value;
      value = getValue(leaf->lexeme);

      return value;

    //or get the a intermediate's value
    }else{

      Value value;
      value.isFunction = 0;
      value.valueType.intValue = ((TOKEN*)tree->left)->value;
      return value;
    }

  }else{

    //get the values from the left and right side of the tree
    int valueLeft = evalExp(tree->left).valueType.intValue;
    int valueRight = evalExp(tree->right).valueType.intValue;
    int finalValue;

    //do operations such as +, -, etc...
    if(tree->type == '+'){

      finalValue = valueLeft + valueRight;

    }else if(tree->type == '-'){

      finalValue = valueLeft - valueRight;

    }else if(tree->type == '*'){

      finalValue = valueLeft * valueRight;

    }else if(tree->type == '/'){

      finalValue = valueLeft / valueRight;

    }else{

      if(valueLeft != 0){

        finalValue = valueLeft;

      }else{

        finalValue = 0;

      }
    }

    Value ret;
    ret.isFunction = 0;
    ret.valueType.intValue = finalValue;

    return ret;
  }

}

//evalulate a condition
int evalCondition(NODE* tree){

  //if there is function call then call it
  if(tree->type == APPLY) return evalFunction(tree).valueType.intValue;

  //evaluate the expression on both the left and right and do the operator on them
  if(tree->type == EQ_OP){

    return evalExp(tree->left).valueType.intValue == evalExp(tree->right).valueType.intValue;

  }else if(tree->type == NE_OP){

    return evalExp(tree->left).valueType.intValue != evalExp(tree->right).valueType.intValue;

  }else if(tree->type == LE_OP){

    return evalExp(tree->left).valueType.intValue <= evalExp(tree->right).valueType.intValue;

  }else if(tree->type == GE_OP){

    return evalExp(tree->left).valueType.intValue >= evalExp(tree->right).valueType.intValue;

  }else if(tree->type == '<'){

    return evalExp(tree->left).valueType.intValue < evalExp(tree->right).valueType.intValue;

  }else if(tree->type == '>'){

    return evalExp(tree->left).valueType.intValue > evalExp(tree->right).valueType.intValue;

  }

  return ((TOKEN*)tree->left)->value;

}
```

6 interpret.h

```
/*
 * interpret.h
 *
 *  Created on: 6 Oct 2016
 *      Author: harry
 */
#include "nodes.h"


#ifndef INTERPRET_H_
#define INTERPRET_H_

/*struct TacLine{
  int variable;
  int operand1;
  int isVar1;
  int operand2;
  int isVar2;
  char operator;
};*/

int interpret(NODE* tree,int level);


void compile(NODE* tree);
void compileToAssembly(NODE* tree,int optimize);

#endif /* INTERPRET_H_ */
```

7 TacLineQueue.c

```
/*
 * TacLineQueue.c
 *
 *  Created on: 19 Oct 2016
 *      Author: harry
 */

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

struct TacLine{

  char* variable;
  int paramType;
  int isVar1Temp;
  char* variable2;
  int isVar2Temp;
  int operand1;
  int isVar1;
  int operand2;
  int isVar2;
  int operator;
  int isSimple;
  int isStatement;
  int isVariableEq;
  int isNext;
  int isRegisterFunctionCall;
  int isVariableCreation;
  int thereIsElse;
  int deleteInOptimization;
  struct TacLine* next;
};


struct TacLine* head = NULL;
int size = 0;

void addToQueue(struct TacLine* next){

  next->deleteInOptimization = 0;
  if(head == NULL){
    head = next;
    size++;

  }else{

    struct TacLine* append = head;
    while(append->isNext == 2){

      append = append->next;
    }

    append->next = next;
    append->isNext = 2;
    size++;
  }
}

struct TacLine* getElement(int index){

  struct TacLine* next = head;
  while(index-- > 0){

    next = next ->next;
  }

  return next;
}


int getSize(){

  return size;
}
```

8 TacLineQueue.h

```
/*
 * TacLineQueue.h
 *
 *  Created on: 19 Oct 2016
 *      Author: harry
 */

#ifndef TACLINEQUEUE_H_
#define TACLINEQUEUE_H_

struct TacLine{

  char* variable;
  int paramType;
  int isVar1Temp;
  char* variable2;
  int isVar2Temp;
  int operand1;
  int isVar1;
  int operand2;
  int isVar2;
  int operator;
  int isSimple;
  int isStatement;
  int isVariableEq;
  int isNext;
  int isRegisterFunctionCall;
  int isVariableCreation;
  int thereIsElse;
  int deleteInOptimization;
  struct TacLine* next;
};

void addToQueue(struct TacLine* next);
struct TacLine* getElement(int index);
int getSize();

#endif /* TACLINEQUEUE_H_ */
```

9 MIPSMemoryInfo.c

```
#include "frames.h"
#include "TacLineQueue.h"
#include <stdio.h>
#include <stdlib.h>


struct Param{

  int number;
  int memoryPos;
  int value;
  struct Param* next;
};

typedef struct Param Param;

Param* head;

void addParam(int value){

  Param* next = (Param*)malloc(sizeof(Param));
  //printf("malloc: %d\n",sizeof(Param));
  next->next = NULL;

  if(head == NULL){

    head = next;

  }else{

    Param* append = head;

    while(append->next != NULL){

      append = append->next;
    }

    append->next = next;
  }

}
```

```
Param* getParam(int no){

  Param* selected = head;
  while(no > 0){

    selected = selected->next;
    no--;
  }

  return selected;
}

struct FunctionInfo{

  int Size;
  char* name;
  int endOfList;
  char* parentFunction;
  struct FunctionInfo* nextFunction;
};

typedef struct FunctionInfo FunctionInfo;

FunctionInfo* newFunctionInfo();

FunctionInfo* funcHead = NULL;
int maxParams;

//calculates the memory allocation for each function
//and the maximum number of parameters
void calculateFunctionInfo(struct TacLine* lines){

  struct TacLine* next = lines;
  FunctionInfo* current;
  maxParams = 0;
  int currentFuncParams = 0;
  int functionCount = 0;

  while(next != NULL){

    if(next->operator == '=' && next->variable[0] != '$'
        && next->isVariableCreation){
      //assignment
      current->Size += 4;

    //argument
    }else if(next->operator == 'A'){

      current->Size += 4;
      currentFuncParams ++;

    //function call
    }else if(next->operator == 'F'){

      current->Size += 8;

    //if and while
    }else if(next->operator == 'C' || next->operator == 'W'){

      current->Size += 4;
      if(next->thereIsElse){

        current->Size += 4;
      }

    //allocate for new function
    }else if(next->operator == 'D'){

      char* context = NULL;
      if(functionCount > 0){

        context = current->name;
      }

      current = newFunctionInfo(context);
      functionCount++;
      current->name = next->variable;
      current->Size = 8;
      //did the last function have the most number of parameters
      if(currentFuncParams > maxParams){

        maxParams = currentFuncParams;
      }
      currentFuncParams = 0;

    }else if(next->operator == 'E'){

      functionCount --;
    }

    next = next->next;
  }

}

//what is the parent of a function
char* getParent(char* functionName){

  FunctionInfo* next = funcHead;

  while(next != NULL){

    if(strcmp(next->name,functionName) == 0){

      return next->parentFunction;
    }

    next = next->nextFunction;
  }

  return NULL;
}

//does the function has a parent
int hasParent(char* functionName){

  return getParent(functionName) != NULL;
}
```

```
int getMaxParams(){

  return maxParams;
}

void printFunctionInfo(){

  FunctionInfo* next = funcHead;

  while(next != NULL){

    if(next == NULL){

      break;
    }

    printf(”Function␣Info:␣%s␣%d␣\n”,next->name,next->Size);

    next = next->nextFunction;
  }
}

//create new structure that hold information about functions
FunctionInfo* newFunctionInfo(char* context){

  FunctionInfo* new = (FunctionInfo*)malloc(sizeof(FunctionInfo));
  new->nextFunction = NULL;
  new->endOfList = 1;
  new->parentFunction = context;

  if(funcHead == NULL){

    funcHead = new;

  }else{

    FunctionInfo* append = funcHead;
    while(!append->endOfList){

      append = append->nextFunction;
    }

    append->endOfList = 0;
    append->nextFunction = new;
  }

  return new;
}


int getBytesToAllocation(char* symbol){

  FunctionInfo* next = funcHead;

  while(1){

    if(next->name == symbol){

      return next->Size;
    }
    if(next->nextFunction == NULL){

      break;
    }

    next = next->nextFunction;
  }

  return 0;
}




Value addNextMemLoc(char* symbol,int isVariableCreation ,int* closureNo){

  if(isVariableCreation){
    Value value = getLastValue();
    value.isFunction = 0;
    value.valueType.intValue += 4;
    addSymbol(symbol,value,isVariableCreation);
    return value;
  }else{
    //need to update values in closure
    return getValueByEquality(symbol,closureNo);
  }


}

int isGlobalFunction(char* symbol){

  return isOnGlobalFrame(symbol);
}

//add the function name to the frame
void addFunctonToFrame(char* functionName){

  Value value;
  value.isFunction = 1;
  value.valueType.intValue = -4;
  addSymbol(functionName,value,1);
}
```

10 MIPSMemoryInfo.h

```
/*
 * MIPSMemoryInfo.h
 *
 *  Created on: 9 Nov 2016
 *      Author: harry
 */

#ifndef MIPSMEMORYINFO_H_
#define MIPSMEMORYINFO_H_

#include ”frames.h”
#include <stdio.h>
```

```
struct Param{

  int number;
  int memoryPos;
  int value;
  struct Param* next;
};
typedef struct Param Param;
void addParam(int value);
Param* getParam(int no);
void calculateFunctionInfo(struct TacLine* lines);
int getMaxParams();
void printFunctionInfo();
Value addNextMemLoc(char* symbol,int isVariableCreation ,int closureNo);
int getBytesToAllocation(char* symbol);
//void setMemoryOffset(int offset);
int hasParent(char* functionName);
char* getParent(char* functionName);
void addFunctonToFrame(char* functionName);
int isGlobalFunction(char* symbol);
#endif /* MIPSMEMORYINFO_H_ */
```

11 main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
//#include "nodes.h"
//#include "C.tab.h"
#include "C.tab.h"
#include "interpret.h"
#include <string.h>

char *named(int t)
{
    static char b[100];
    if (isgraph(t) || t=='␣') {
      sprintf(b, "%c", t);
      return b;
    }
    switch (t) {
      default: return "???";
    case IDENTIFIER:
      return "id";
    case CONSTANT:
      return "constant";
    case STRING_LITERAL:
      return "string";
    case LE_OP:
      return "<=";
    case GE_OP:
      return ">=";
    case EQ_OP:
      return "==";
    case NE_OP:
      return "!=";
    case EXTERN:
      return "extern";
    case AUTO:
      return "auto";
    case INT:
      return "int";
    case VOID:
      return "void";
    case APPLY:
      return "apply";
    case LEAF:
      return "leaf";
    case IF:
      return "if";
    case ELSE:
      return "else";
    case WHILE:
      return "while";
    case CONTINUE:
      return "continue";
    case BREAK:
      return "break";
    case RETURN:
      return "return";
    }
}

void print_leaf(NODE *tree, int level)
{
  //printf("Type leaf: %c \n",named(tree->type));
    TOKEN *t = (TOKEN *)tree;
    int i;
    for(i=0; i<level; i++) putchar('␣');//putchar(i+'1');
    if (t->type == CONSTANT) printf("Value:␣%d\n", t->value);
    else if (t->type == STRING_LITERAL) printf("STRING_LITERAL:␣\"%s\"\n", t->lexeme);
    else if (t){

      printf("Puts:␣%s␣Type:␣%s\n",t->lexeme,named(tree->type));
      //puts(t->lexeme);

    }
}

void print_tree0(NODE *tree, int level)
{
    int i;
    if (tree==NULL) return;
    if (tree->type==LEAF) {
      print_leaf(tree->left, level);
    }
    else {
      for(i=0; i<level; i++) putchar('␣');//putchar(i+'1');
      printf("Type:␣%s\n", named(tree->type));
/*       if (tree->type=='~') { */
/*         for(i=0; i<level+2; i++) putchar(' '); */
/*         printf("%p\n", tree->left); */
/*       } */
/*       else */

        //for(i=0; i<level; i++) putchar(i+'1');
        //printf("LEFT: %s\n", named(tree->type));
        print_tree0(tree->left, level+2);
        //for(i=0; i<level; i++) putchar(i+'1');
        //printf("RIGHT: %s\n", named(tree->type));
        print_tree0(tree->right, level+2);
    }
}

void print_tree(NODE *tree)
{
    print_tree0(tree, 0);
}


extern int yydebug;
extern NODE* yyparse(void);
extern NODE* ans;
extern void init_symbtable(void);

int main(int argc, char** argv)
{
    NODE* tree;
    if(argc < 2){

      printf("Too␣few␣arguments");
    }
    char* option = argv[1];
    if (argc>2 && strcmp(argv[2],"-d")==0) yydebug = 1;
    init_symbtable();

    yyparse();
    tree = ans;

    printf("\n\n");

    if(strcmp(option,"INT") == 0){

      printf("—–C␣INTEPRETER\n");
      printf("Answer:␣%d\n",interpret(tree,0));

    }else if(strcmp(option,"CMP") == 0){

      printf("—–C␣COMPILER\n");
      compileToAssembly(tree,0);

    }else if(strcmp(option, "TAC") == 0){

      printf("—–C␣TAC␣ONLY␣COMPILER\n");
      compile(tree);

    }else if(strcmp(option,"CMPOPT") == 0){

      printf("—–C␣OPTIMIZE␣COMPILER\n");
      compileToAssembly(tree,1);

    }else if(strcmp(option,"TREE") == 0){

      printf("PRINT␣TREE\n");
      print_tree(tree);

    }else{

      printf("Invalid␣option.");
    }
    return 0;
}
```

12 MakeFile

```
OBJS = lex.yy.o C.tab.o symbol_table.o nodes.o main.o interpret.o hash.o frames.o compile.o TacL
SRCS = lex.yy.c C.tab.c symbol_table.c nodes.c main.c interpret.c hash.c frames.c compile.c TacL
CC = gcc

all:   mycc

clean:
  rm ${OBJS}

mycc: ${OBJS}
  ${CC} -g -o mycc ${OBJS}

lex.yy.c: C.flex
  flex C.flex

C.tab.c:   C.y
  bison -d -t -v C.y

.c.o:
  ${CC} -g -c $*.c

depend:
  ${CC} -M $(SRCS) > .deps
  cat Makefile .deps > makefile

dist: symbol_table.c nodes.c main.c interpret.c Makefile C.flex C.y nodes.h token.h interpret.h
  tar cvfz mycc.tgz symbol_table.c nodes.c main.c interpret.c hash.c frames.c compile.c MIPSMem
    nodes.h token.h interpret.h
```