```
#ifndef VALUE_
#define VALUE_

typedef struct frame FRAME;

typedef struct closure {
FRAME* env;
NODE* code;
} CLOSURE;

typedef struct value {
    int type;
    union
    {
        int integer;
        int boolean;
        char* string;
        CLOSURE* closure;

    } ;

}VALUE;

typedef struct valuelist {
    VALUE *value;
    struct valuelist *next;
}VALUELIST;

#endif
```

```
#include "token.h"

#ifndef MCENV
#define MCENV
#define MAXREGS 8
#define MAXARGS 4

typedef struct tac TAC;
typedef struct frme FRME;

typedef struct clsure {
FRME* env;
TAC* code;
int processed;
} CLSURE;

typedef struct bnding {
  TOKEN* name;
  int type;
  union {TOKEN* loc; CLSURE* clos;};
  struct bnding* next;
} BNDING;

typedef struct frme {
  BNDING* bindings;
  int size;
  int stack_pos;
  struct frme* next;
}FRME;

TOKEN *lookup_loc(TOKEN*, FRME*);
TOKEN *assign_to_var(TOKEN*, FRME*,TOKEN*);
void declare_var(TOKEN*, FRME*);
int reg_in_use(int, FRME*);
void delete_constants(FRME*);
TOKEN* use_temp_reg(FRME *);
#endif
```

```
1 #include "nodes.h"
2
3 void print_tree(NODE *tree);
```

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

#ifndef INTERPRETER
#define INTERPRETER

VALUE* interpret_tree(NODE*,FRAME*);
VALUE* make_value_int(int);


#endif // INTERPRETER
```

```
#include "token.h"
#include "nodes.h"
#include "mc_env.h"
#ifndef GENTAC
#define GENTAC

typedef struct env {
  int dstcounter;
  int lblcounter;
  TOKEN* currlbl;
}ENV;

enum tac_op
  {
    tac_plus = 1,
    tac_minus = 2,
    tac_div = 3,
    tac_mod = 4,
    tac_mult = 5,

    tac_proc = 6,
    tac_endproc = 7,
    tac_load = 8,
    tac_store = 9,
    tac_if = 10,
    tac_lbl = 11,
    tac_goto = 12,
    tac_call = 13,
    tac_rtn = 14,
    tac_innerproc = 15
  };

typedef struct simple_tac {
  TOKEN* src1;
  TOKEN* src2;
  TOKEN* dst;
}STAC;

typedef struct proc {
  TOKEN* name;
  int arity;
  TOKENLIST* args;
}PROC;

typedef struct load {
  TOKEN* src1;
  TOKEN* dst;
}LOAD;

typedef struct label {
  TOKEN* name;
}LABEL;

typedef struct iftest {
  TOKEN* op1;
  TOKEN* op2;
  int code;
  TOKEN* lbl;
}IFTEST;
```

```
61 typedef struct gotolbl {
62   TOKEN* lbl;
63 }GOTO;
64
65 typedef struct call {
66  TOKEN* name;
67  int arity;
68  TOKENLIST* args;
69 } CALL;
70
71 typedef struct rtn {
72   int type;
73   union {CALL call; TOKEN* v;};
74 }RTN;
75
76 typedef struct tac {
77 int op ;
78 union {STAC stac; PROC proc; LOAD ld; LABEL lbl; IFTEST ift; GOTO gtl; CALL
   call; RTN rtn;};
79 struct tac* next;
80 }TAC;
81
82 typedef struct bb {
83   TOKEN* id;
84   TAC* leader;
85   TAC* end;
86   struct bb *nexts[2];
87 }BB;
88
89 TAC*gen_tac(NODE*);
90 TAC* gen_tac0(NODE*, ENV*,FRME*,int);
91
92
93 #endif
```

```c
#include "gentac.h"

#ifndef GENMC
#define GENMC


enum{
  PRINT_INT = 1,
  PRINT_CHAR = 11,
  SBRK = 9,
  EXIT = 10
};

typedef struct mc {
  char* insn;
  struct mc* next;
} MC;

typedef struct var {
  TOKEN* name;
  TOKEN* reg;
  struct var* next;
}VAR;

typedef struct sr {
  int srcnt;
}SR;

typedef struct ar {
int size;
int arity;
unsigned int pc; // save caller ’s PC
unsigned int sl; // this function ’s static link
//unsigned int param[MAXARGS]; // param0 , ... paramm ,
unsigned int local[MAXREGS]; // local0 , ... localn ,
//unsigned int tmp[k];
} AR;

MC* gen_mc(TAC*);
MC* gen_mc0(TAC*, FRME*, AR*);
TOKEN * new_dst(FRME *);
#endif
```

```
#include "nodes.h"
#include "value.h"
#ifndef ENVIRONMENT
#define ENVIRONMENT

typedef struct binding {
  TOKEN* name;
  VALUE* value;
  struct binding* next;
} BINDING;

typedef struct frame {
  BINDING* bindings;
  struct frame* next;
}FRAME;


extern VALUE *lookup_name(TOKEN*, FRAME*);
extern VALUE *assign_to_name(TOKEN*, FRAME*,VALUE*);
extern VALUE *declare_name(TOKEN*, FRAME*);
#endif //ENVIRONMENT
```

```c
#include "token.h"
#include <stdlib.h>
#include "regstack.h"

#define MAXREGSIZE 8
#define MAXARGS 4
#define MAXLBLS 10

TOKEN* stack[MAXREGSIZE];
TOKEN* arg_stack[MAXARGS];
TOKEN* lbl_stack[MAXLBLS];

int top = -1;
int top_args = -1;
int top_lbls = -1;

int isempty() {

   if(top == -1)
      return 1;
   else
      return 0;
}

int isempty_args() {

   if(top_args == -1)
      return 1;
   else
      return 0;
}

int isempty_lbls() {

   if(top_lbls == -1)
      return 1;
   else
      return 0;
}

int isfull() {

   if(top == MAXREGSIZE)
      return 1;
   else
      return 0;
}

int isfull_args() {

   if(top_args == MAXARGS)
      return 1;
   else
      return 0;
}

int isfull_lbls() {

   if(top_lbls == MAXLBLS)
      return 1;
```

```
61    else
62       return 0;
63 }
64 
65 TOKEN* pop() {
66    TOKEN* data;
67 
68    if(!isempty()) {
69       data = stack[top];
70       top = top - 1;
71       return data;
72    } else {
73       return NULL;
74    }
75 }
76 
77 TOKEN* peep(){
78     TOKEN* data;
79     if(!isempty()) {
80       data = stack[top];
81       return data;
82    } else {
83       return NULL;
84    }
85 }
86 
87 TOKEN* peep_lbl(){
88     TOKEN* data;
89     if(!isempty_lbls()) {
90       data = lbl_stack[top_lbls];
91       return data;
92    } else {
93       return NULL;
94    }
95 }
96 
97 int push(TOKEN* data) {
98 
99    if(!isfull()) {
100      top = top + 1;
101      stack[top] = data;
102      return 0;
103   } else {
104      return -1;
105   }
106 }
107 
108 int push_arg(TOKEN* data) {
109 
110 if(!isfull_args()) {
111   top_args = top_args + 1;
112   arg_stack[top_args] = data;
113   return 0;
114 } else {
115   return -1;
116 }
117 }
118 
119 TOKEN* pop_arg() {
120   TOKEN* data;
```

```
121
122    if(!isempty_args()) {
123       data = arg_stack[top_args];
124       top_args = top_args - 1;
125       return data;
126    } else {
127       return NULL;
128    }
129 }
130
131 int push_lbl(TOKEN* data) {
132
133 if(!isfull_lbls()) {
134    top_lbls = top_lbls + 1;
135    lbl_stack[top_lbls] = data;
136    return 0;
137 } else {
138    return -1;
139 }
140 }
141
142 TOKEN* pop_lbl() {
143    TOKEN* data;
144
145    if(!isempty_lbls()) {
146       data = lbl_stack[top_lbls];
147       top_lbls = top_lbls - 1;
148       return data;
149    } else {
150       return NULL;
151    }
152 }
```

```
#include <stdlib.h>
#include <stdio.h>
#include "mc_env.h"
#include "string.h"
#include "C.tab.h"
#include "genmc.h"

extern TOKEN * new_dst(FRME *);

TOKEN *lookup_loc(TOKEN * x, FRME * frame){
    while(frame != NULL){
        BNDING *bindings = frame->bindings;
        while(bindings != NULL){
            if(bindings->name == x){
                return bindings->loc;
            }
            bindings = bindings->next;
        }
        return NULL;
    }
}

TOKEN* lookup_reg(int x, FRME * frame){
    while(frame != NULL){
        BNDING *bindings = frame->bindings;
        while(bindings != NULL){
            if(bindings->loc != NULL && bindings->loc->value == x){
                return bindings->name;
            }
            bindings = bindings->next;
        }
        return NULL;
    }
}

void delete_loc(TOKEN * x, FRME * frame){
    while(frame != NULL){
        BNDING *bindings = frame->bindings;
        BNDING *head = bindings;
        BNDING *prev = NULL;
        while(bindings != NULL){
            if(bindings->name == x){
                if(prev == NULL){
                    frame->bindings = bindings->next;
                }
                else{
                    prev->next = bindings->next;
                    frame->bindings = head;
                }
                return;
            }
            prev = bindings;
            bindings = bindings->next;
        }
        frame = frame->next;
    }
}

void delete_constants(FRME* frame){
     while(frame != NULL){
```

```
61          BNDING *bindings = frame->bindings;
62          while(bindings != NULL){
63              if(bindings->name->type == CONSTANT){
64                  delete_loc(bindings->name,frame);
65              }
66              bindings = bindings->next;
67          }
68          frame = frame->next;
69      }
70  }
71
72  int reg_in_use(int x, FRME * frame){
73      while(frame != NULL){
74          BNDING *bindings = frame->bindings;
75          while(bindings != NULL){
76              if(bindings->loc != NULL && bindings->loc->value == x){
77                  return 1;
78              }
79              bindings = bindings->next;
80          }
81          frame = frame->next;
82      }
83      return 0;
84  }
85
86  TOKEN* use_temp_reg(FRME * frame){
87      TOKEN* t= new_dst(frame);
88      if(t == NULL ) {printf("error: all registers in use!");exit(1);}
89      BNDING *bindings = frame->bindings;
90      BNDING *new = malloc(sizeof(BNDING));
91      if(new != NULL){
92          new->type = IDENTIFIER;
93          new->loc = t;
94          new->next = bindings;
95          frame->bindings=new;
96          return t;
97      }
98      printf("fatal: binding creation failed!\n");
99  }
100
101 TOKEN *assign_to_var(TOKEN * x, FRME * frame, TOKEN* loc){
102     while(frame != NULL){
103         BNDING *bindings = frame->bindings;
104         while(bindings != NULL){
105             if(bindings->name == x){
106                 if(reg_in_use(loc->value,frame)){
107                     delete_loc(lookup_reg(loc->value,frame),frame);
108                 }
109                 bindings->loc = loc;
110                 return loc;
111             }
112             bindings = bindings->next;
113         }
114         frame = frame->next;
115     }
116     printf("fatal: unbound variable!\n");exit(1);
117 }
118
119 void declare_var(TOKEN * x, FRME * frame){
120     BNDING *bindings = frame->bindings;
```

```
121     BNDING *new = malloc(sizeof(BNDING));
122     if(new != NULL){
123         new->type = IDENTIFIER;
124         new->name = x;
125         new->loc = NULL;
126         new->next = bindings;
127         frame->bindings=new;
128         return;
129     }
130     printf("fatal: binding creation failed!\n");
131 }
132 
133 TOKEN *declare_fnc(TOKEN * x, CLSURE* val, FRME * frame){
134     BNDING *bindings = frame->bindings;
135     BNDING *new = malloc(sizeof(BNDING));
136     if(new != NULL){
137         new->type = CLOS;
138         new->name = x;
139         new->clos = val;
140         new->next = bindings;
141         frame->bindings=new;
142         return new->name;
143     }
144     printf("fatal: binding creation failed!\n");
145 }
146 
147 CLSURE *find_fnc(TOKEN* name, FRME* e){
148     FRME *ef = e;
149     BNDING* bindings;
150      while(ef != NULL){
151         bindings = ef->bindings;
152         while (bindings != NULL){
153             if(bindings->name ==  name){
154                return bindings->clos;
155             }
156             bindings = bindings->next;
157         }
158          ef = ef->next;
159      }
160      return NULL;
161 }
162 
163
```

```c
#include <stdio.h>
#include <ctype.h>
#include "nodes.h"
#include "C.tab.h"
#include <string.h>
#include "interpreter.h"
#include "gentac.h"
#include "genmc.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)
```

```
 61 {
 62     TOKEN *t = (TOKEN *)tree;
 63     int i;
 64     for (i=0; i<level; i++) putchar(' ');
 65     if (t->type == CONSTANT) printf("%d\n", t->value);
 66     else if (t->type == STRING_LITERAL) printf("\"%s\"\n", t->lexeme);
 67     else if (t) puts(t->lexeme);
 68 }
 69
 70 void print_tree0(NODE *tree, int level)
 71 {
 72     int i;
 73     if (tree==NULL) return;
 74     if (tree->type==LEAF) {
 75       print_leaf(tree->left, level);
 76     }
 77     else {
 78       for(i=0; i<level; i++) putchar(' ');
 79       printf("%s\n",named(tree->type));
 80 /*       if (tree->type=='~') { */
 81 /*         for(i=0; i<level+2; i++) putchar(' '); */
 82 /*         printf("%p\n", tree->left); */
 83 /*       } */
 84 /*       else */
 85         print_tree0(tree->left, level+2);
 86       print_tree0(tree->right, level+2);
 87     }
 88 }
 89
 90 void print_tree(NODE *tree)
 91 {
 92     print_tree0(tree, 0);
 93 }
 94
 95 char* tac_ops[] =
    {"","ADD","SUB","DIV","MOD","MULT","PROC","ENDPROC","LOAD","STORE","IF","LABEL
    ","GOTO","CALL","RETURN","INNER_PROC"};
 96
 97 void print_if(TAC* tac){
 98   if(tac->ift.op1->type == IDENTIFIER && tac->ift.op2->type == IDENTIFIER){
 99     printf("%s (%s%s%s) %s\n",
100     tac_ops[tac->op],
101     tac->ift.op1->lexeme,
102     named(tac->ift.code),
103     tac->ift.op2->lexeme,
104     tac->ift.lbl->lexeme);
105   }
106   else if(tac->ift.op1->type == IDENTIFIER){
107     printf("%s (%s%s%d) %s\n",
108     tac_ops[tac->op],
109     tac->ift.op1->lexeme,
110     named(tac->ift.code),
111     tac->ift.op2->value,
112     tac->ift.lbl->lexeme);
113   }
114   else if(tac->ift.op2->type == IDENTIFIER){
115     printf("%s (%d%s%s) %s\n",
116     tac_ops[tac->op],
117     tac->ift.op1->value,
118     named(tac->ift.code),
```

```
119     tac->ift.op2->lexeme,
120     tac->ift.lbl->lexeme);
121   }
122   else{
123     printf("%s (%d%s%d) %s\n",
124     tac_ops[tac->op],
125     tac->ift.op1->value,
126     named(tac->ift.code),
127     tac->ift.op2->value,
128     tac->ift.lbl->lexeme);
129   }
130
131 }
132
133 void print_rtn(TAC* tac){
134   if(tac->rtn.type == tac_call){
135     printf("%s\n",
136     tac_ops[tac->op]);
137   }
138   else if (tac->rtn.type == CONSTANT){
139     printf("%s %i\n",
140     tac_ops[tac->op],
141     tac->rtn.v->value);
142   }
143   else{
144     printf("%s %s\n",
145     tac_ops[tac->op],
146     tac->rtn.v->lexeme);
147   }
148 }
149
150 void print_ic(TAC* tac){
151
152   while(tac!=NULL){
153     switch(tac->op){
154       default:
155         printf("%s %s %s %s\n",
156         tac_ops[tac->op],
157         tac->stac.src1->lexeme,
158         tac->stac.src2->lexeme,
159         tac->stac.dst->lexeme);
160         break;
161       case tac_load:
162         if(tac->ld.src1->type == CONSTANT){
163           printf("%s %i %s\n",
164           tac_ops[tac->op],
165           tac->ld.src1->value,
166           tac->ld.dst->lexeme);
167         }
168         else{
169           printf("%s %s %s\n",
170           tac_ops[tac->op],
171           tac->ld.src1->lexeme,
172           tac->ld.dst->lexeme);
173         }
174         break;
175       case tac_store:
176         printf("%s %s %s\n",
177         tac_ops[tac->op],
178         tac->ld.src1->lexeme,
```

```
179         tac->ld.dst->lexeme);
180         break;
181       case tac_proc:
182         printf("%s %s %i\n",
183         tac_ops[tac->op],
184         tac->proc.name->lexeme,
185         tac->proc.arity);
186         break;
187       case tac_innerproc:
188         printf("%s %s %i\n",
189         tac_ops[tac->op],
190         tac->proc.name->lexeme,
191         tac->proc.arity);
192         break;
193       case tac_endproc:
194         printf("%s\n",
195         tac_ops[tac->op]);
196         break;
197       case tac_if:
198         print_if(tac);
199         break;
200       case tac_lbl:
201         printf("%s %s\n",
202         tac_ops[tac->op],
203         tac->lbl.name->lexeme);
204         break;
205       case tac_goto:
206         printf("%s %s\n",
207         tac_ops[tac->op],
208         tac->gtl.lbl->lexeme);
209         break;
210       case tac_call:
211         printf("%s %s %i\n",
212         tac_ops[tac->op],
213         tac->call.name->lexeme,
214         tac->call.arity);
215         break;
216       case tac_rtn:
217         print_rtn(tac);
218         break;
219     }
220     tac = tac->next;
221   }
222
223 }
224
225 void print_token(TOKEN *i){
226   if (i->type == CONSTANT){
227     printf("%d",i->value);
228   }
229   else {
230     printf("%s",i->lexeme);
231   }
232 }
233 void print_mc(MC* i)
234 {
235   for(;i!=NULL;i=i->next) printf("%s\n",i->insn);
236 }
237
238 void print_bbs(BB** bbs){
```

```
239   int i = 0;
240   while(bbs[i] != NULL){
241     printf("\033[0;31m");
242     printf("BLOCK #");print_token(bbs[i]->id);
243     printf("\n\033[0m");
244     print_ic(bbs[i]->leader);
245     if(bbs[i]->nexts[0] != NULL){
246       printf("\033[0;31m");
247       printf("LINKS TO : ");
248       print_token(bbs[i]->nexts[0]->id);
249     }
250     if(bbs[i]->nexts[1] != NULL){
251       printf(" ");
252       print_token(bbs[i]->nexts[1]->id);
253     }
254     i++;
255     printf("\n\n");
256   }
257   printf("\033[0m");
258 }
259
260 extern int yydebug;
261 extern NODE* yyparse(void);
262 extern NODE* ans;
263 extern void init_symbtable(void);
264 extern VALUE* interpret(NODE*);
265 extern TAC* gen_tac(NODE*);
266 extern MC* gen_mc(TAC*);
267
268 int main(int argc, char** argv)
269 {
270     NODE* tree;
271     FRAME* e = malloc(sizeof(FRAME));
272     if (argc>1 && strcmp(argv[1],"-d")==0) yydebug = 1;
273     init_symbtable();
274     printf("--C COMPILER\n");
275     yyparse();
276     tree = ans;
277     printf("parse finished with %p\n", tree);
278     print_tree(tree);
279     printf("\n");
280     printf("Calling interpreter...\n");
281     VALUE* result = interpret(tree);
282     if(result != NULL){
283       printf("RESULT : %i\n",result->integer);
284     }
285     else{
286       printf("RESULT: NULL\n");
287     }
288
289
    printf("-----------------------------------------------------------------\n");
290     printf("Generating TAC...\n");
291     TAC* tac = gen_tac(tree);
292     print_ic(tac);
293     //print_bbs(tac);
294     printf("Generating machine code...\n");
295     print_mc(gen_mc(tac));
296     return 0;
297 }
```

298

```c
#include "nodes.h"
#include "interpreter.h"
#include "environment.h"
#include "main.h"
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "C.tab.h"
#include "value.h"

extern NODE *tree;
extern void print_tree(NODE *tree);
extern VALUE *lookup_name(TOKEN*, FRAME*);
extern VALUE *lookup_name_curr_frame(TOKEN*, FRAME*);
extern VALUE *assign_to_name(TOKEN*, FRAME*,VALUE*);
extern VALUE *declare_name(TOKEN*, FRAME*);
extern VALUE *declare_func(TOKEN*,VALUE*, FRAME*);

int r_early = 0, in_seq = 0;

//built-ins

void print_int(VALUE *v){
    if(v->type == CONSTANT){
         printf("%d\n", v->integer);
    }
    else{
        printf("fatal: print_int(): invalid int type\n"); exit(1);
    }
}

void print_string(VALUE *v){
    if(v->type == STRING_LITERAL){
        printf("%s\n",v->string);
    }
    else{
        printf("fatal: print_string(): invalid string type\n"); exit(1);
    }
}

VALUE* read_int(){
    int n;
    int c;
    char buf[2];
    printf("> ");
    clearerr(stdin);
    fgets(buf, 2,stdin);
    n = strtol(buf, NULL,10);
    return make_value_int(n);
}

int is_builtin(TOKEN* name){
    char *ps = "print_string";
    char *p = "print_int";
    char *r = "read_int";


if(!strcmp(ps,name->lexeme)||!strcmp(p,name->lexeme)||!strcmp(r,name->lexeme))
{
        return 1;
```

```
59      }
60      return 0;
61  }
62
63  VALUE* call_builtin(TOKEN* name, VALUELIST* args){
64      char *ps = "print_string";
65      char *p = "print_int";
66      char *r = "read_int";
67
68      if(!strcmp(ps,name->lexeme)){
69          print_string(args->value);
70          return NULL;
71      }
72      if(!strcmp(p,name->lexeme)){
73          print_int(args->value);
74          return NULL;
75      }
76      if(!strcmp(r,name->lexeme)){
77          return read_int();
78      }
79  }
80
81  VALUE* new_closure(NODE* t, FRAME* e){
82      CLOSURE* c = malloc(sizeof(CLOSURE));
83      VALUE* v = malloc(sizeof(VALUE));
84      if(c == NULL || v == NULL){printf("fatal: cannot allocate memory for
    closure\n");exit(1);}
85      c->code=t;
86      c->env=e;
87      v->type = CLOS;
88      v->closure = c;
89      return v;
90  }
91
92  TOKENLIST* find_tokens(NODE* ids){
93      TOKENLIST* tokens = malloc(sizeof(TOKENLIST));
94      if((char)ids->type == '~'){
95          tokens->name = (TOKEN*)ids->right->left;
96          return tokens;
97      }
98      else{
99          if((char)ids->type == ','){
100             tokens->name = (TOKEN*)ids->right->right->left;
101             tokens->next = find_tokens(ids->left);
102             return tokens;
103         }
104     }
105 }
106
107 FRAME *extend_frame(FRAME* e, NODE *ids, VALUELIST *args){
108
109     FRAME* new_frame = malloc(sizeof(FRAME));
110     if(ids == NULL && args == NULL) {return new_frame;}
111     BINDING *bindings = NULL;
112     new_frame->bindings = bindings;
113     //while (ids != NULL && args != NULL) {
114        TOKENLIST* tokens = find_tokens(ids);
115        while(tokens != NULL && args != NULL){
116             declare_name(tokens->name,new_frame);
117             assign_to_name(tokens->name,new_frame,args->value);
```

```
118             tokens=tokens->next;
119             args = args->next;
120         }
121         if(!(tokens == NULL && args == NULL)){
122             printf("error: invalid number of arguments and/or tokens,
    exiting...\n");exit(1);
123         }
124     return new_frame;
125 }
126
127 VALUE* make_value_int(int val){
128     VALUE *value = malloc(sizeof(VALUE));
129     if (value == NULL) {perror("fatal: make_value_int failed\n"); exit(1);}
130
131     value->type = CONSTANT;
132     value->integer = val;
133     return value;
134 }
135
136 VALUE* make_value_bool(int val){
137     VALUE *value = malloc(sizeof(VALUE));
138     if (value == NULL) {perror("fatal: make_value_bool failed\n"); exit(1);}
139
140     value->type = BOOL;
141     value->boolean = val;
142     return value;
143 }
144
145 VALUE* make_value_string(char* str){
146     VALUE *value = malloc(sizeof(VALUE));
147     if (value == NULL) {perror("fatal: make_value_string failed\n"); exit(1);}
148
149     value->type = STRING_LITERAL;
150     value->string = malloc(strlen(str));
151     strcpy(value->string, str);
152     return value;
153 }
154
155 VALUE* interpret_tilde(NODE*tree, FRAME* e){
156     TOKEN* t;
157     if(tree->left->left->type==INT || tree->left->left->type==FUNCTION ||
    tree->left->left->type==STRING_LITERAL){
158         if(tree->right->type == LEAF){
159             t = (TOKEN *)tree->right->left;
160             if(lookup_name(t,e) == NULL){return declare_name(t,e);}
161             else {printf("error: multiple declarations of
    %s",t->lexeme);exit(1);}
162         }
163         else if((char)tree->right->type == '='){
164                 t = (TOKEN *)tree->right->left->left;
165             if(lookup_name_curr_frame(t,e) == NULL){declare_name(t,e);}
166             else {printf("error: multiple declarations of variable
    '%s'\n",t->lexeme);exit(1);}
167             return assign_to_name(t,e,interpret_tree(tree->right->right,e));
168         }
169     }
170     interpret_tree(tree->left,e);
171     return interpret_tree(tree->right,e);
172 }
173
```

```
174 VALUE* if_method(NODE* tree, FRAME* e){
175     VALUE* condition = interpret_tree(tree->left,e);
176     if(tree->right->type == ELSE){
177         NODE* consequent = tree->right->left;
178         NODE* alternative = tree->right->right;
179         if(condition->type == BOOL){
180             if(condition->boolean){
181                 return interpret_tree(consequent,e);
182             }
183             else{
184                 return interpret_tree(alternative,e);
185             }
186         }
187         else{printf("error: condition is not boolean value\n");exit(1);}
188     }
189     else{
190         NODE* consequent = tree->right;
191         if(condition->type == BOOL){
192             if(condition->boolean){
193                 return interpret_tree(consequent,e);
194             }
195         }
196         else{printf("error: condition is not boolean value\n");exit(1);}
197     }
198 }
199
200 CLOSURE *find_func(TOKEN* name, FRAME* e){
201     FRAME *ef = e;
202     BINDING* bindings;
203      while(ef != NULL){
204         bindings = ef->bindings;
205         while (bindings != NULL){
206             if(bindings->name ==  name){
207                return bindings->value->closure;
208             }
209             bindings = bindings->next;
210         }
211          ef = ef->next;
212      }
213      printf("No function %s in scope, exiting...\n",name->lexeme);exit(1);
214 }
215
216 NODE* formals(CLOSURE* f){
217     return f->code->left->right->right;
218 }
219
220 VALUE *call(NODE* name, FRAME* e, VALUELIST* args){
221     TOKEN* t = (TOKEN *)name;
222     if(is_builtin(t)){
223         return call_builtin(t,args);
224     }
225     CLOSURE *f = find_func(t,e);
226     FRAME* ef = extend_frame(e,formals(f),args);
227     ef->next = f->env;
228     return interpret_tree(f->code->right,ef);
229 }
230
231 VALUELIST* find_curr_values(NODE *t, FRAME* e){
232     VALUELIST *values = malloc(sizeof(VALUELIST));
233     if(t == NULL) return NULL;
```

```
234     char c = (char)t->type;
235     if(t->type == LEAF || c == '*' || c == '+' || c == '-' || c == '%'|| c ==
    '/' || t->type == APPLY){
236         values->value = interpret_tree(t,e);
237         values->next = NULL;
238         return values;
239     }
240     else if((char)t->type == ','){
241             values->value = interpret_tree(t->right,e);
242             values->next = find_curr_values(t->left,e);
243             return values;
244     }
245     else{
246         printf("fatal: invalid parameter in call.\n");exit(1);
247     }
248 }
249
250 VALUE* interpret(NODE* tree){
251     FRAME* e = malloc(sizeof(FRAME));
252     interpret_tree(tree,e);
253     FRAME *ef = e;
254     while(ef != NULL){
255         BINDING* bindings = e->bindings;
256         while (bindings != NULL){
257             if(strcmp(bindings->name->lexeme,"main")==0){
258                 return call(bindings->name,e,NULL);
259             }
260             bindings = bindings->next;
261         }
262         ef = e->next;
263     }
264     printf("No main function. exiting...\n");exit(1);
265
266 }
267
268 VALUE* interpret_tree(NODE *tree, FRAME* e){
269
270     VALUE *left, *right;
271     TOKEN *t;
272
273     if (tree==NULL) {printf("fatal: no tree received\n") ; exit(1);}
274     if (tree->type==LEAF){
275         t = (TOKEN *)tree->left;
276         if (t->type == CONSTANT){
277             return make_value_int(t->value);
278         }
279         else if (t->type == IDENTIFIER){
280             VALUE *v = lookup_name(t,e);
281             if (v==NULL){
282                 printf("error: undefined variable %s\n",t->lexeme);
283             }
284             else{return v;}
285         }
286         else if (t->type == STRING_LITERAL){
287             return make_value_string(t->lexeme);
288         }
289     }
290     char c = (char)tree->type;
291     if (isgraph(c) || c==' ') {
292         switch(c){
```

```
293             default: printf("fatal: unknown token type '%c'\n",c); exit(1);
294 
295             case '~':
296                 return interpret_tilde(tree,e);
297             case 'D':
298             //case 'd':
299                 t = (TOKEN *)tree->left->right->left->left;
300                 return declare_func(t,new_closure(tree,e),e);
301             case ';':
302                 in_seq = 1;
303                 if(tree->left != NULL){
304                     left = interpret_tree(tree->left,e); //HOW DO YOU STOP
    EXECUTING BELOW IF THIS RETURNS ??
305                     if(r_early){
306                         return left;
307                     }
308                 }
309                 in_seq = 0;
310                 return interpret_tree(tree->right,e);
311             case '=':
312                 interpret_tree(tree->left,e);
313                 t = (TOKEN *)tree->left->left;
314                 return assign_to_name(t,e,interpret_tree(tree->right,e));
315             case '+':
316                 left = interpret_tree(tree->left,e);
317                 right = interpret_tree(tree->right,e);
318                 return make_value_int(left->integer + right->integer);
319             case '-':
320                 left = interpret_tree(tree->left,e);
321                 right = interpret_tree(tree->right,e);
322                 return make_value_int(left->integer - right->integer);
323             case '*':
324                 left = interpret_tree(tree->left,e);
325                 right = interpret_tree(tree->right,e);
326                 return make_value_int(left->integer * right->integer);
327             case '/':
328                 left = interpret_tree(tree->left,e);
329                 right = interpret_tree(tree->right,e);
330                 return make_value_int(left->integer / right->integer);
331             case '%':
332                 left = interpret_tree(tree->left,e);
333                 right = interpret_tree(tree->right,e);
334                 return make_value_int(left->integer % right->integer);
335             case '>':
336                 if(interpret_tree(tree->left,e)->integer >
    interpret_tree(tree->right,e)->integer){
337                     return make_value_bool(1);
338                 }
339                 else{return make_value_bool(0);}
340             case '<':
341                 if(interpret_tree(tree->left,e)->integer <
    interpret_tree(tree->right,e)->integer){
342                     return make_value_bool(1);
343                 }
344                 else{return make_value_bool(0);}
345         }
346     }
347     switch(tree->type){
348         default: printf("fatal: unknown token type '%i'\n", tree->type);
    exit(1);
```

```
349         case RETURN:
350             if(in_seq){
351                 r_early = 1;
352             }
353             return interpret_tree(tree->left,e);
354         case IF:
355             return if_method(tree,e);
356         case APPLY:
357             return call(tree->left->left,e,find_curr_values(tree->right,e));
358         case LE_OP:
359             if(interpret_tree(tree->left,e)->integer <=
    interpret_tree(tree->right,e)->integer){
360                 return make_value_bool(1);
361             }
362             else{return make_value_bool(0);}
363         case GE_OP:
364         if(interpret_tree(tree->left,e)->integer >=
    interpret_tree(tree->right,e)->integer){
365                 return make_value_bool(1);
366             }
367             else{return make_value_bool(0);}
368         case EQ_OP:
369             if(interpret_tree(tree->left,e)->integer ==
    interpret_tree(tree->right,e)->integer){
370                     return make_value_bool(1);
371                 }
372                 else{return make_value_bool(0);}
373         case NE_OP:
374             if(interpret_tree(tree->left,e)->integer !=
    interpret_tree(tree->right,e)->integer){
375                     return make_value_bool(1);
376                 }
377                 else{return make_value_bool(0);}
378     }
379 }
```

```
#include "gentac.h"
#include <stdlib.h>
#include "C.tab.h"
#include "nodes.h"
#include <stdio.h>
#include <ctype.h>
#include "value.h"
#include "mc_env.h"
#include "token.h"
#include "string.h"
#include "regstack.h"
#include "hashtable.h"

extern TOKEN* new_token(int);
extern int isempty() ;
extern int isfull() ;
extern TOKEN* pop();
extern TOKEN* pop_arg();
extern TOKEN* peep();
extern int push(TOKEN*);
extern int push_arg(TOKEN*);
extern BB* insert(TOKEN*,TAC*);

extern TOKEN *lookup_loc(TOKEN*, FRME*);
extern TOKEN *assign_to_var(TOKEN*, FRME*,TOKEN*);
extern void declare_var(TOKEN*, FRME*);
extern int reg_in_use(int, FRME*);
extern void delete_constants(FRME*);


TOKEN* new_lbl(ENV *env){
    TOKEN* lbl = (TOKEN*)malloc(sizeof(TOKEN));
    if(lbl==NULL){printf("fatal: failed to generate destination\n");exit(1);}
    lbl->type=IDENTIFIER;
    lbl->lexeme = (char*)calloc(1,2);
    sprintf(lbl->lexeme,"L%i",env->lblcounter);
    lbl->value = env->lblcounter;
    env->lblcounter++;
    env->currlbl = lbl;
    return lbl;
}

TOKEN * new_dest(FRME *e){
    for(int i=0; i<MAXREGS; i++){
        if(!reg_in_use(i,e)){
            TOKEN* dst = (TOKEN*)malloc(sizeof(TOKEN));
            if(dst==NULL){printf("fatal: failed to generate
destination\n");exit(1);}
            dst->type=IDENTIFIER;
            dst->lexeme = (char*)calloc(1,2);
            sprintf(dst->lexeme,"t%i",i);
            dst->value = i;
            return dst;
        }
    }
}


TAC* find_last(TAC* tac){
    while(tac->next!=NULL){
```

```
60          tac = tac->next;
61      }
62      return tac;
63  }
64
65  TOKEN* find_last_dest(TAC* tac){
66      tac = find_last(tac);
67      switch (tac->op){
68          case tac_plus:
69          case tac_minus:
70          case tac_div:
71          case tac_mod:
72          case tac_mult:
73              return tac->stac.dst;
74
75          case tac_load:
76              return tac->ld.dst;
77
78          case tac_store:
79              return tac->ld.src1;
80      }
81  }
82
83  ENV *init_env(){
84      ENV *env = malloc(sizeof(ENV));
85      if (env==NULL) {
86          printf("Error! memory not allocated.");
87          exit(0);
88      }
89      env->lblcounter=0;
90      new_lbl(env);
91      return env;
92  }
93
94  TAC* empty_tac() {
95      TAC* ans = (TAC*)malloc(sizeof(TAC));
96      if (ans==NULL) {
97          printf("Error! memory not allocated.");
98          exit(0);
99      }
100     return ans;
101 }
102
103
104 TAC* new_stac(int op, TOKEN* src1, TOKEN* src2, TOKEN* dst){
105   TAC* ans = empty_tac();
106   ans->op = op;
107   ans->stac.src1 = src1;
108   ans->stac.src2 = src2;
109   ans->stac.dst = dst;
110   return ans;
111 }
112
113 TAC* new_proc (TOKEN* name, int arity, TOKENLIST* args){
114     TAC* ans = empty_tac();
115     ans->op = tac_proc;
116     ans->proc.name = name;
117     ans->proc.arity = arity;
118     ans->proc.args = args;
119     return ans;
```

```
120 }
121
122 TAC* new_innerproc (TOKEN* name, int arity, TOKENLIST* args){
123     TAC* ans = empty_tac();
124     ans->op = tac_innerproc;
125     ans->proc.name = name;
126     ans->proc.arity = arity;
127     ans->proc.args = args;
128     return ans;
129 }
130
131 TAC* new_load(TOKEN* name, FRME* e){
132     TAC* ans = empty_tac();
133     ans->op = tac_load;
134     ans->ld.src1 = name;
135     TOKEN* t = lookup_loc(name,e);
136     if(t == NULL){
137         t = new_dest(e);
138         declare_var(name,e);
139         assign_to_var(name,e,t);
140     }
141     ans->ld.dst = t;
142     return ans;
143 }
144
145 TAC* new_store(TOKEN* name, TOKEN* dst, FRME *e,ENV* env){
146     TAC* ans = empty_tac();
147     ans->op = tac_store;
148     ans->ld.dst = dst;
149     TOKEN* t = lookup_loc(dst,e);
150     if(t == NULL){
151         declare_var(dst,e);
152     }
153     assign_to_var(dst,e,name);
154     ans->ld.src1 = name;
155     return ans;
156 }
157
158
159 int count_params(NODE * tree){
160     int count = 0;
161     if (tree == NULL || tree->type == INT || tree->type == FUNCTION ||
    tree->type == STRING_LITERAL) {return 0;}
162     if( tree->type == LEAF && tree->left->type==IDENTIFIER){
163         return 1;
164     }
165     else{
166         count += count_params(tree->left);
167         count += count_params(tree->right);
168         return count;
169     }
170 }
171
172 TOKENLIST* get_params(NODE* ids){
173     if(ids == NULL){return NULL;}
174     TOKENLIST* tokens = malloc(sizeof(TOKENLIST));
175     if((char)ids->type == '~'){
176         tokens->name = (TOKEN*)ids->right->left;
177         return tokens;
178     }
```

```
179     else{
180         if((char)ids->type == ','){
181             tokens->name = (TOKEN*)ids->right->right->left;
182             tokens->next = get_params(ids->left);
183             return tokens;
184         }
185     }
186 }
187
188 TAC* new_endproc(){
189     TAC* ans = empty_tac();
190     ans->op = tac_endproc;
191     return ans;
192 }
193
194 TAC* new_if(TOKEN* op1, TOKEN* op2, int code, TOKEN* lbl){
195     TAC* ans = empty_tac();
196     ans->op= tac_if;
197     ans->ift.code = code;
198     ans->ift.op1 = op1;
199     ans->ift.op2 = op2;
200     ans->ift.lbl = lbl;
201     return ans;
202 }
203
204 TAC* new_goto(TOKEN* lbl){
205     TAC* ans = empty_tac();
206     ans->op= tac_goto;
207     ans->gtl.lbl = lbl;
208     return ans;
209 }
210
211 TAC* new_label(TOKEN* lbl){
212     TAC* ans = empty_tac();
213     ans->op= tac_lbl;
214     ans->lbl.name = lbl;
215     return ans;
216 }
217
218 TAC* parse_tilde(NODE* tree, FRME* e, ENV* env, int depth){
219     TAC *tac,*last;
220     TOKEN* t;
221      if(tree->left->left->type==INT){
222         if(tree->right->type == LEAF){
223             t = (TOKEN *)tree->right->left;
224             TOKEN* new = new_token(CONSTANT);
225             TOKEN* reg = new_dest(e);
226             new->value = 0;
227             tac = new_load(new,e);
228             tac->next = new_store(reg,t,e,env);
229             return tac;
230         }
231         else if((char)tree->right->type == '='){
232             t = (TOKEN *)tree->right->left->left;
233             tac = gen_tac0(tree->right->right,env,e,depth);
234             last = find_last(tac);
235             if(last->stac.dst != NULL){
236                 last->next = new_store(last->stac.dst,t,e,env);
237             }
238             else{  last->next = new_store(last->ld.dst,t,e,env); }
```

```
239             return tac;
240         }
241     }
242     tac = gen_tac0(tree->left,env,e,depth);
243     last = find_last(tac);
244     last->next = gen_tac0(tree->right,env,e,depth);
245     return tac;
246 }
247
248 TAC* parse_if(NODE* tree, ENV* env, FRME *e, int depth){
249     int code = tree->left->type;
250     TOKEN* op1 = (TOKEN*)tree->left->left->left;
251     TOKEN* op2 = (TOKEN*)tree->left->right->left;
252     TAC* last1,*last2;
253     new_lbl(env);
254     TAC* tacif = new_if(op1,op2,code,env->currlbl);
255     if(tree->right->type == ELSE){
256         TAC* consequent = gen_tac0(tree->right->left,env,e,depth);
257         TAC* alternative = gen_tac0(tree->right->right,env,e,depth);
258         TAC* altlbl = new_label(env->currlbl);
259         new_lbl(env);
260         TAC* gtl = new_goto(env->currlbl);
261
262         last1 = find_last(alternative);
263         last1->next = new_label(env->currlbl);
264         altlbl->next = alternative;
265         gtl->next = altlbl;
266         last2 = find_last(consequent);
267         last2->next = gtl;
268         tacif->next = consequent;
269         return tacif;
270     }
271     else{
272         TAC* consequent = gen_tac0(tree->right,env,e,depth);
273         consequent->next = new_label(env->currlbl);
274         tacif->next = consequent;
275         return tacif;
276     }
277 }
278
279 int count_args(NODE * tree){
280     int count = 0;
281     if (tree == NULL) {return 0;}
282     if( tree->type == LEAF){
283         return 1;
284     }
285     else{
286         count += count_args(tree->left);
287         count += count_args(tree->right);
288         return count;
289     }
290 }
291
292 TOKENLIST* get_args(NODE *tree, ENV* env, FRME* e){
293     TOKENLIST* tokens = malloc(sizeof(TOKENLIST));
294     if(tree == NULL){return NULL;}
295     char c = (char)tree->type;
296     if(tree->type == LEAF){
297         tokens->name = (TOKEN*)tree->left;
298         return tokens;
```

```
299     }
300     else{
301         if((char)tree->type == ','){
302             tokens->name = (TOKEN*)tree->right->left;
303             tokens->next = get_args(tree->left,env,e);
304             return tokens;
305         }
306     }
307 }
308
309 TAC* new_call(NODE* tree, ENV* env, FRME* e){
310     TAC* ans = empty_tac();
311     ans->op = tac_call;
312     ans->call.name = (TOKEN*)tree->left->left;
313     ans->call.arity = count_args(tree->right);
314     ans->call.args = get_args(tree->right,env,e);
315     return ans;
316 }
317
318 TAC* new_return(NODE* tree, ENV* env, FRME* e, int depth){
319     TAC* ans = empty_tac();
320     TAC* last;
321     ans->op = tac_rtn;
322     if (tree->type==LEAF){
323         TOKEN *t = (TOKEN *)tree->left;
324         ans->rtn.type = t->type;
325         ans->rtn.v = t;
326     }
327     else if (tree->type==APPLY){
328        ans = new_call(tree,env,e);
329        last = find_last(ans);
330        last->next = empty_tac();
331        last->next->op = tac_rtn;
332        last->next->rtn.type = tac_call;
333     }
334     else{
335         TAC* tac = gen_tac0(tree,env,e,depth);
336         TOKEN* t = find_last_dest(tac);
337         TAC* last = find_last(tac);
338         ans->rtn.type = t->type;
339         ans->rtn.v = t;
340         last->next = ans;
341         return tac;
342     }
343     delete_constants(e);
344     return ans;
345 }
346
347 TAC *gen_tac0(NODE *tree, ENV* env, FRME* e, int depth){
348
349     TOKEN *left = malloc(sizeof(TOKEN)), *right = malloc(sizeof(TOKEN));
350     TAC *tac, *last;
351     TOKEN *t;
352
353     if (tree==NULL) {printf("fatal: no tree received\n") ; exit(1);}
354     if (tree->type==LEAF){
355             t = (TOKEN *)tree->left;
356             tac = new_load(t,e);
357             return tac;
358         }
```

```
359     char c = (char)tree->type;
360     if (isgraph(c) || c==' ') {
361         switch(c){
362             default: printf("fatal: unknown token type '%d'\n",c); exit(1);
363 
364             case '~':
365                return parse_tilde(tree,e,env,depth);
366             case 'D':
367                 tac = gen_tac0(tree->left,env,e,++depth);
368                 last = find_last(tac);
369                 last->next = gen_tac0(tree->right,env,e,++depth);
370                 last = find_last(tac);
371                 last->next = new_endproc();
372                 return tac;
373             case 'd':
374                 return gen_tac0(tree->right,env,e,depth);
375             case 'F':
376                 left = (TOKEN*)tree->left->left;
377                 if(depth > 1){
378                     return
    new_innerproc(left,count_params(tree->right),get_params(tree->right));
379                 }
380                 else{
381                     return
    new_proc(left,count_params(tree->right),get_params(tree->right));
382                 }
383 
384             case ';':
385                 tac = gen_tac0(tree->left,env,e,depth);
386                 last = find_last(tac);
387                 last->next = gen_tac0(tree->right,env,e,depth);
388                 return tac;
389             case '=':
390                 tac = gen_tac0(tree->right,env,e,depth);
391                 last = find_last(tac);
392                 t = (TOKEN *)tree->left->left;
393                 if(last->stac.dst != NULL){
394                     last->next = new_store(last->stac.dst,t,e,env);
395                 }
396                 else if(last->op = tac_call){
397                     last->next = new_store(new_dest(e),t,e,env);
398                 }
399                 else{  last->next = new_store(last->ld.dst,t,e,env); }
400                 delete_constants(e);
401                 return tac;
402             case '+':
403                 tac = gen_tac0(tree->left,env,e,depth);
404                 left = find_last_dest(tac);
405                 last = find_last(tac);
406                 last->next = gen_tac0(tree->right,env,e,depth);
407                 right = find_last_dest(last->next);
408                 last = find_last(last);
409                 t = new_token(CONSTANT);
410                 declare_var(t,e);
411                 assign_to_var(t,e,new_dest(e));
412                 last->next = new_stac(tac_plus,left,right,lookup_loc(t,e));
413                 return tac;
414             case '-':
415                 tac = gen_tac0(tree->left,env,e,depth);
416                 left = find_last_dest(tac);
```

```
417                 last = find_last(tac);
418                 last->next = gen_tac0(tree->right,env,e,depth);
419                 right = find_last_dest(last->next);
420                 last = find_last(last);
421                 t = new_token(CONSTANT);
422                 declare_var(t,e);
423                 assign_to_var(t,e,new_dest(e));
424                 last->next = new_stac(tac_minus,left,right,lookup_loc(t,e));
425                 return tac;
426             case '*':
427                 tac = gen_tac0(tree->left,env,e,depth);
428                 left = find_last_dest(tac);
429                 last = find_last(tac);
430                 last->next = gen_tac0(tree->right,env,e,depth);
431                 right = find_last_dest(last->next);
432                 last = find_last(last);
433                 t = new_token(CONSTANT);
434                 declare_var(t,e);
435                 assign_to_var(t,e,new_dest(e));
436                 last->next = new_stac(tac_mult,left,right,lookup_loc(t,e));
437                 return tac;
438             case '/':
439                 tac = gen_tac0(tree->left,env,e,depth);
440                 left = find_last_dest(tac);
441                 last = find_last(tac);
442                 last->next = gen_tac0(tree->right,env,e,depth);
443                 right = find_last_dest(last->next);
444                 last = find_last(last);
445                 t = new_token(CONSTANT);
446                 declare_var(t,e);
447                 assign_to_var(t,e,new_dest(e));
448                 last->next = new_stac(tac_div,left,right,lookup_loc(t,e));
449                 return tac;
450             case '%':
451                 tac = gen_tac0(tree->left,env,e,depth);
452                 left = find_last_dest(tac);
453                 last = find_last(tac);
454                 last->next = gen_tac0(tree->right,env,e,depth);
455                 right = find_last_dest(last->next);
456                 last = find_last(last);
457                 t = new_token(CONSTANT);
458                 declare_var(t,e);
459                 assign_to_var(t,e,new_dest(e));
460                 last->next = new_stac(tac_mod,left,right,lookup_loc(t,e));
461                 return tac;
462         }
463     }
464     switch(tree->type){
465     default: printf("fatal: unknown token type '%c'\n", tree->type); exit(1);
466     case RETURN:
467         return new_return(tree->left,env,e,depth);
468     case IF:
469         return parse_if(tree,env,e,depth);
470     case APPLY:
471         return new_call(tree,env,e);
472     }
473 }
474
475 TAC* find_in_seq(TAC* seq, TAC* target){
476     while(seq!=target){
```

```
477         seq = seq->next;
478     }
479     return seq;
480 }
481
482 BB* find_bb(BB** bbs, TOKEN* id, int size){
483
484     for(int i=0; i<size; i++){
485         if(bbs[i] != NULL && bbs[i]->id == id){
486             return bbs[i];
487         }
488     }
489     return NULL;
490 }
491
492 BB* find_next_bb(BB** bbs, TOKEN* id, int size){
493     for(int i=0; i<size; i++){
494         if(bbs[i] != NULL && bbs[i]->id->value == (id->value+1)){
495             return bbs[i];
496         }
497     }
498     return NULL;
499 }
500
501 BB** gen_bbs(TAC* tac){
502     static BB* bbs[10];
503     //bb->nexts = malloc(sizeof(BB)*2);
504     TAC *curr;
505     int i =0;
506     int id = 0;
507     while(tac != NULL){
508         BB* bb = malloc(sizeof(BB));
509         bb->leader = tac;
510         curr = tac->next;
511         while(curr->op != tac_if && curr->op != tac_goto && curr->next != NULL
    && curr->next->op != tac_lbl){
512             curr = curr->next;
513         }
514         tac = curr->next;
515         curr = find_in_seq(bb->leader,curr);
516         /* switch(curr->op){
517             case tac_if:
518                 bb->nexts[0] = gen_bbs(tac);
519                 bb->nexts[1] = insert(curr->ift.lbl,NULL);
520                 break;
521             case tac_goto:
522                 bb->nexts[0] = insert(curr->gtl.lbl,NULL);
523                 break;
524         }  */
525         curr->next = NULL;
526         bbs[i] = bb;
527         i++;
528
529         if(bb->leader->op == tac_lbl){
530             bb->id = bb->leader->lbl.name;
531         }
532         else{
533             TOKEN* c = new_token(CONSTANT); c->value = id;
534             bb->id = c;
535             id++;
```

```
536         }
537     }
538     TAC* transfer;
539     i = 0;
540     while(bbs[i] != NULL){
541         transfer = find_last(bbs[i]->leader);
542         if(transfer->op == tac_goto){
543             bbs[i]->nexts[0] = find_bb(bbs,transfer->gtl.lbl,10);
544         }
545         else{
546             bbs[i]->nexts[0] = find_next_bb(bbs,bbs[i]->id,10);
547             if(transfer->op == tac_if){
548                 bbs[i]->nexts[1] = find_bb(bbs,transfer->ift.lbl,10);
549             }
550         }
551         i++;
552     }
553     return bbs;
554 }
555
556 TAC *gen_tac(NODE* tree){
557     ENV *env = init_env();
558     FRME* e = malloc(sizeof(FRME));
559     TAC* tac = gen_tac0(tree,env,e,0);
560     //BB** bbs = gen_bbs(tac);
561     return tac;
562 }
```

```c
#include "genmc.h"
#include "gentac.h"
#include <stdlib.h>
#include <stdio.h>
#include "C.tab.h"
#include "regstack.h"
#include "mc_env.h"
#include "string.h"

#define INSN_BUF 64

extern TOKEN *lookup_loc(TOKEN*, FRME*);
extern TOKEN *assign_to_var(TOKEN*, FRME*,TOKEN*);
extern void declare_var(TOKEN*, FRME*);
extern void declare_fnc(TOKEN*, CLSURE*, FRME*);
extern CLSURE *find_fnc(TOKEN* , FRME* );
extern TOKEN* use_temp_reg(FRME *);

int call_stack;

TAC* find_endproc(TAC* i){
  int nested_depth = 0;
  while(i != NULL){
    if(i->op == tac_innerproc){
      nested_depth++;
    }
    if(i->op == tac_endproc){
      if(nested_depth == 0){
        return i;
      }
      else{
        nested_depth--;
      }
    }
    i = i->next;
  }
  return i;
}

MC* find_lst(MC* mc){
  while(mc->next != NULL){
    mc = mc->next;
  }
  return mc;
}

int count_locals(TAC* i){
  int n = 0;
  while(i != NULL && i->op != tac_endproc){
    if(i->op == tac_store){
      n++;
    }
    i = i->next;
  }
  return n;
}

TOKEN * new_dst(FRME *e){
    for(int i=0; i<MAXREGS; i++){
        if(!reg_in_use(i,e)){
```

```
 61             TOKEN* dst = (TOKEN*)malloc(sizeof(TOKEN));
 62             if(dst==NULL){printf("fatal: failed to generate
    destination\n");exit(1);}
 63             dst->type=IDENTIFIER;
 64             dst->lexeme = (char*)calloc(1,2);
 65             sprintf(dst->lexeme,"t%i",i);
 66             dst->value = i;
 67             return dst;
 68         }
 69     }
 70 }
 71
 72 MC* new_minus(TAC* tac){
 73   MC *mc = malloc(sizeof(MC));
 74   mc->insn = malloc(sizeof(INSN_BUF));
 75   sprintf(mc->insn,"sub
    $%s,$%s,$%s",tac->stac.dst->lexeme,tac->stac.src1->lexeme,tac->stac.src2->lexe
    me);
 76   return mc;
 77 }
 78 MC* new_div(TAC* tac){
 79   MC *mc = malloc(sizeof(MC));
 80   mc->insn = malloc(sizeof(INSN_BUF));
 81   sprintf(mc->insn,"div
    $%s,$%s",tac->stac.src1->lexeme,tac->stac.src2->lexeme);
 82   mc->next = malloc(sizeof(MC));
 83   mc->next->insn = malloc(sizeof(INSN_BUF));
 84   sprintf(mc->next->insn,"mflo $%s",tac->stac.dst->lexeme);
 85   return mc;
 86 }
 87 MC* new_mod(TAC* tac){
 88   MC *mc = malloc(sizeof(MC));
 89   mc->insn = malloc(sizeof(INSN_BUF));
 90   sprintf(mc->insn,"div
    $%s,$%s",tac->stac.src1->lexeme,tac->stac.src2->lexeme);
 91   mc->next = malloc(sizeof(MC));
 92   mc->next->insn = malloc(sizeof(INSN_BUF));
 93   sprintf(mc->next->insn,"mfhi $%s",tac->stac.dst->lexeme);
 94   return mc;
 95 }
 96 MC* new_mult(TAC* tac){
 97   MC *mc = malloc(sizeof(MC));
 98   mc->insn = malloc(sizeof(INSN_BUF));
 99   sprintf(mc->insn,"mult
    $%s,$%s",tac->stac.src1->lexeme,tac->stac.src2->lexeme);
100   mc->next = malloc(sizeof(MC));
101   mc->next->insn = malloc(sizeof(INSN_BUF));
102   sprintf(mc->next->insn,"mflo $%s",tac->stac.dst->lexeme);
103   return mc;
104 }
105 MC* new_plus(TAC* tac){
106   MC *mc = malloc(sizeof(MC));
107   mc->insn = malloc(sizeof(INSN_BUF));
108   sprintf(mc->insn,"add
    $%s,$%s,$%s",tac->stac.dst->lexeme,tac->stac.src1->lexeme,tac->stac.src2->lexe
    me);
109   return mc;
110 }
111
112 MC* init_mc(){
```

```
113     MC *mc = malloc(sizeof(MC));
114     mc->insn = malloc(sizeof(INSN_BUF));
115     mc->insn = ".globl main";
116     mc->next = malloc(sizeof(MC));
117     mc->next->insn = malloc(sizeof(INSN_BUF));
118     mc->next->insn = ".text";
119     return mc;
120 }
121
122 MC* make_syscall(int code){
123   MC* mc;
124   mc = malloc(sizeof(MC));
125   mc->insn = malloc(sizeof(INSN_BUF));
126   sprintf(mc->insn,"li $v0 %d",code);
127
128   MC* last = find_lst(mc);
129   last->next =  malloc(sizeof(MC));
130   last->next->insn = malloc(sizeof(INSN_BUF));
131   last->next->insn = "syscall";
132   return mc;
133 }
134
135 MC* new_smpl_ld(FRME* e, TOKEN* src, TOKEN* dst){
136   MC *mc = malloc(sizeof(MC));
137     mc->insn = malloc(sizeof(INSN_BUF));
138     if(src->type == CONSTANT){
139         sprintf(mc->insn, "li $%s,%d",dst->lexeme,src->value);
140     }
141     else if(src->type == IDENTIFIER){
142         TOKEN *loc = lookup_loc(src,e);
143         sprintf(mc->insn, "move $%s,$%s",dst->lexeme,src->lexeme);
144     }
145     return mc;
146 }
147
148 TOKEN* lookup_from_memory(TOKEN* name, FRME* e, AR* ar){
149     MC *mc = malloc(sizeof(MC));
150     mc->insn = malloc(sizeof(INSN_BUF));
151     mc->insn = "# Looking up token from memory";
152     MC* last = find_lst(mc);
153     int j = 0;
154     TOKEN* t;
155    while(e != NULL){
156     BNDING *bindings = e->bindings;
157     int i = 1;
158     while(bindings != NULL){
159         if(bindings->name == name){
160           t = new_token(IDENTIFIER);
161           t->lexeme = malloc(sizeof(INSN_BUF));
162           sprintf(t->lexeme,"%d($sp)",call_stack+ar->size-e->stack_pos-8-4*i);
163           return t;
164         }
165         if(bindings->type == IDENTIFIER){
166           i++;
167         }
168         bindings = bindings->next;
169     }
170     j+= e->size;
171     e = e->next;
172   }
```

```
173   return t;
174 }
175
176 MC* new_ld(FRME *e, TAC* tac, AR* curr){
177     MC *mc = malloc(sizeof(MC));
178     mc->insn = malloc(sizeof(INSN_BUF));
179     if(tac->ld.src1->type == CONSTANT){
180         sprintf(mc->insn, "li
    $%s,%d",tac->ld.dst->lexeme,tac->ld.src1->value);
181     }
182     else if(tac->ld.src1->type == IDENTIFIER){
183         TOKEN *loc = lookup_loc(tac->ld.src1,e);
184         if(loc == NULL){
185           loc = lookup_from_memory(tac->ld.src1,e,curr);
186           sprintf(mc->insn, "lw $%s, %s",tac->ld.dst->lexeme,loc->lexeme);
187         }
188         else{
189           sprintf(mc->insn, "move $%s,$%s",tac->ld.dst->lexeme,loc->lexeme);
190         }
191     }
192     return mc;
193 }
194
195  MC* new_str(TAC* tac, FRME* e){
196   MC *mc = malloc(sizeof(MC));
197   MC *last;
198   mc->insn = malloc(sizeof(INSN_BUF));
199   TOKEN* t = lookup_loc(tac->ld.dst,e);
200   if(t == NULL){
201     declare_var(tac->ld.dst,e);
202     assign_to_var(tac->ld.dst,e,tac->ld.src1);
203   }
204   else if(t->value != tac->ld.src1->value) {
205     assign_to_var(tac->ld.dst,e,tac->ld.src1);
206   }
207   return mc;
208 }
209
210 MC* new_ift(TAC* tac, FRME* e){
211   TOKEN* dst1 = lookup_loc(tac->ift.op1,e);
212   MC* mc = NULL;
213   if(dst1 == NULL){
214     dst1 = new_dst(e);
215     declare_var(tac->ift.op1,e);
216     assign_to_var(tac->ift.op1,e,dst1);
217     mc = new_smpl_ld(e,tac->ift.op1,dst1);
218   }
219   TOKEN* dst2 = lookup_loc(tac->ift.op2,e);
220   if(dst2 == NULL){
221     dst2 = new_dst(e);
222     declare_var(tac->ift.op2,e);
223     assign_to_var(tac->ift.op2,e,dst2);
224     if(mc != NULL){
225       mc->next = new_smpl_ld(e,tac->ift.op2,dst2);
226     }
227     else {mc = new_smpl_ld(e,tac->ift.op2,dst2);}
228   }
229
230   MC* last = find_lst(mc);
231   if(last != NULL){
```

```
232     last->next = malloc(sizeof(MC));
233     last->next->insn = malloc(sizeof(INSN_BUF));
234     last = last->next;
235   }
236   else{
237     last = malloc(sizeof(MC));
238     last->insn = malloc(sizeof(INSN_BUF));
239   }
240
241   switch(tac->ift.code){
242     case '>':
243       sprintf(last->insn,"ble $%s $%s
    %s",dst1->lexeme,dst2->lexeme,tac->ift.lbl->lexeme);
244       break;
245     case '<':
246       sprintf(last->insn,"bge $%s $%s
    %s",dst1->lexeme,dst2->lexeme,tac->ift.lbl->lexeme);
247       break;
248     case EQ_OP:
249       sprintf(last->insn,"bne $%s $%s
    %s",dst1->lexeme,dst2->lexeme,tac->ift.lbl->lexeme);
250       break;
251     case NE_OP:
252       sprintf(last->insn,"beq $%s $%s
    %s",dst1->lexeme,dst2->lexeme,tac->ift.lbl->lexeme);
253       break;
254     case LE_OP:
255       sprintf(last->insn,"bgt $%s $%s
    %s",dst1->lexeme,dst2->lexeme,tac->ift.lbl->lexeme);
256       break;
257     case GE_OP:
258       sprintf(last->insn,"blt $%s $%s
    %s",dst1->lexeme,dst2->lexeme,tac->ift.lbl->lexeme);
259   }
260   delete_constants(e);
261   return mc;
262 }
263
264 MC* new_gtl(TAC* i){
265   MC* mc = malloc(sizeof(MC));
266   mc->insn = malloc(sizeof(INSN_BUF));
267   sprintf(mc->insn,"j %s",i->gtl.lbl->lexeme);
268   return mc;
269 }
270
271 MC* new_lbli(TAC* i){
272   MC* mc = malloc(sizeof(MC));
273   mc->insn = malloc(sizeof(INSN_BUF));
274   sprintf(mc->insn,"%s:",i->lbl.name->lexeme);
275   return mc;
276 }
277
278 MC* save_frame(AR* ar, FRME *e){
279   MC *mc = malloc(sizeof(MC));
280   mc->insn = malloc(sizeof(INSN_BUF));
281   mc->insn = "# Saving frame";
282   MC* last = find_lst(mc);
283   int i = 0;
284   while(e != NULL && ar->arity != 0){
285     BNDING *bindings = e->bindings;
```

```
286     int i = 1;
287     while(bindings != NULL){
288         last->next = malloc(sizeof(MC));
289         last->next->insn = malloc(sizeof(INSN_BUF));
290         if(bindings->type == IDENTIFIER){
291           sprintf(last->next->insn,"sw $%s
    %d($sp)",bindings->loc->lexeme,8+4*i);
292            i++;
293         }
294 
295         bindings = bindings->next;
296         last = find_lst(last);
297     }
298     break;
299   }
300   last->next = malloc(sizeof(MC));
301   last->next->insn = malloc(sizeof(INSN_BUF));
302   last->next->insn = "# End of saving frame";
303   return mc;
304 }
305 
306 MC* gen_frame(AR* ar){
307 
308 
309   MC *mc = malloc(sizeof(MC));
310   mc->insn = malloc(sizeof(INSN_BUF));
311   mc->insn = "# Creating new frame";
312   MC* last = find_lst(mc);
313 
314   //allocate stack space for new frame
315   last->next = malloc(sizeof(MC));
316   last->next->insn = malloc(sizeof(INSN_BUF));
317   sprintf(last->next->insn,"addiu $sp, $sp -%d",ar->size);
318   last = find_lst(mc);
319 
320   //load return address
321   last->next = malloc(sizeof(MC));
322   last->next->insn = malloc(sizeof(INSN_BUF));
323   sprintf(last->next->insn,"sw $ra, 4($sp)");
324   last = find_lst(mc);
325 
326   //load new size into reg
327   last->next = malloc(sizeof(MC));
328   last->next->insn = malloc(sizeof(INSN_BUF));
329   sprintf(last->next->insn,"li $t1, %d",ar->size);
330   last = find_lst(mc);
331 
332    //store size on stack
333   last->next = malloc(sizeof(MC));
334   last->next->insn = malloc(sizeof(INSN_BUF));
335   last->next->insn = "sw $t1, 0($sp)";
336   last = find_lst(mc);
337 
338   last->next = malloc(sizeof(MC));
339   last->next->insn = malloc(sizeof(INSN_BUF));
340   last->next->insn = "# End of creating frame";
341   return mc;
342 }
343 
344 MC* gen_globframe(TAC* tac, FRME* e, AR* global){
```

```
345   global->sl = 0;
346   global->size = 12;
347   global->arity = 0;
348   MC *mc = malloc(sizeof(MC));
349   mc->insn = malloc(sizeof(INSN_BUF));
350   mc->insn = "#saving global frame";
351   MC* last = find_lst(mc);
352   CLSURE* f;
353   while(tac != NULL){
354       switch(tac->op){
355         case(tac_load):
356            last->next = new_ld(e,tac,global);
357            last = find_lst(last);
358            last->next = new_str(tac->next,e);
359            global->size+=4;
360            global->arity++;
361            break;
362         case(tac_proc):
363           f = malloc(sizeof(CLSURE));
364           f->env = e;
365           f->code = tac;
366           f->processed = 0;
367           declare_fnc(tac->proc.name,f,e);
368           tac = find_endproc(tac);
369       }
370       tac = tac->next;
371   }
372   e->size = global->size;
373   MC* first =malloc(sizeof(MC));
374   first->insn = malloc(sizeof(INSN_BUF));
375   first->insn = "main: ";
376   MC* r = find_lst(first);
377   r->next = gen_frame(global);
378   r = find_lst(r);
379   r->next = mc;
380   r = find_lst(r);
381   r->next = save_frame(global,e);
382   return first;
383 }
384
385 AR* calculate_frame(AR* old, TAC* tac){
386   AR* new = malloc(sizeof(AR));
387   int locals = count_locals(tac);
388   new->arity = locals + tac->proc.arity;
389   new->size = (locals*4) + (tac->proc.arity*4)+12;
390   new->sl = old->sl+1;
391   return new;
392 }
393
394 MC* restore_frame(AR* ar, FRME *e){
395   MC *mc = malloc(sizeof(MC));
396   mc->insn = malloc(sizeof(INSN_BUF));
397   mc->insn = "# Restoring frame";
398   MC* last = find_lst(mc);
399   int i = 0;
400   while(e != NULL && ar->arity != 0){
401     BNDING *bindings = e->bindings;
402     int i = 1;
403     while(bindings != NULL){
404         last->next = malloc(sizeof(MC));
```

```
405         last->next->insn = malloc(sizeof(INSN_BUF));
406         if(bindings->type == IDENTIFIER){
407           sprintf(last->next->insn,"lw $%s
    %d($sp)",bindings->loc->lexeme,8+4*i);
408         }
409         i++;
410         bindings = bindings->next;
411         last = find_lst(last);
412     }
413     break;
414   }
415   //restore return address
416   last->next = malloc(sizeof(MC));
417   last->next->insn = malloc(sizeof(INSN_BUF));
418   last->next->insn = "lw $ra 4($sp)";
419   last = find_lst(last);
420
421   last->next = malloc(sizeof(MC));
422   last->next->insn = malloc(sizeof(INSN_BUF));
423   last->next->insn = "# End of restoring frame";
424   return mc;
425 }
426
427 FRME *extend_frme(FRME* e, TAC *ids, TOKENLIST *args){
428
429     FRME* new_frame = malloc(sizeof(FRME));
430     if(ids == NULL && args == NULL) {return new_frame;}
431     BNDING *bindings = NULL;
432     new_frame->bindings = bindings;
433     //while (ids != NULL && args != NULL) {
434        TOKENLIST* tokens = ids->proc.args;
435        TOKEN* loc;
436        while(tokens != NULL && args != NULL){
437             declare_var(tokens->name,new_frame);
438             assign_to_var(tokens->name,new_frame,new_dst(new_frame));
439             tokens=tokens->next;
440             args = args->next;
441        }
442        if(!(tokens == NULL && args == NULL)){
443            printf("error: invalid number of arguments and/or tokens,
    exiting...\n");exit(1);
444        }
445     return new_frame;
446 }
447
448
449 MC *call_func(TOKEN* name, TAC* call, FRME* e, AR* curr){
450   TOKEN* t = (TOKEN *)name;
451   CLSURE *f = find_fnc(t,e);
452   MC *mc = malloc(sizeof(MC));
453   mc->insn = malloc(sizeof(INSN_BUF));
454   if(!f->processed){
455     f->processed = 1;
456     FRME* ef;
457     if(call != NULL){
458       ef = extend_frme(e,f->code,call->call.args);
459     }
460     else{
461       ef = extend_frme(e,f->code,NULL);
462     }
```

```
463     ef->next = f->env;
464     call_stack += curr->size;
465     ef->stack_pos = call_stack;
466     mc = gen_mc0(f->code,ef,curr);
467   }
468    return mc;
469 }
470
471
472 MC* new_func_rtn(TAC* i){
473   MC *mc = malloc(sizeof(MC));
474   mc->insn = malloc(sizeof(INSN_BUF));
475   if(i->next == NULL){
476   }
477   else{
478     mc->insn = "jr $ra";
479   }
480   return mc;
481 }
482
483 MC* new_rtn(TAC* tac, FRME* e, AR* ar){
484     MC *mc = malloc(sizeof(MC));
485     mc->insn = malloc(sizeof(INSN_BUF));
486     if(tac->rtn.type == CONSTANT){
487       sprintf(mc->insn,"li $v1 %d",tac->rtn.v->value);
488     }
489     else if(tac->rtn.type == IDENTIFIER){
490       TOKEN *t = lookup_loc(tac->rtn.v,e);
491       if(t == NULL){
492         t = lookup_from_memory(tac->rtn.v,e,ar);
493       }
494       if(t == NULL){
495         sprintf(mc->insn,"move $v1 $%s",tac->rtn.v->lexeme);
496       }
497       else{
498         sprintf(mc->insn,"move $v1 $%s",t->lexeme);
499       }
500     }
501     call_stack -= e->size;
502     MC* last = find_lst(mc);
503     last->next =  malloc(sizeof(MC));
504     last->next->insn = malloc(sizeof(INSN_BUF));
505     sprintf(last->next->insn,"addiu $sp, $sp %d",ar->size);
506     last = find_lst(last);
507     last->next =  malloc(sizeof(MC));
508     last->next->insn = malloc(sizeof(INSN_BUF));
509     sprintf(last->next->insn,"jr $ra");
510     return mc;
511 }
512
513 MC* new_prc(TAC* tac, FRME* e){
514   MC *mc = malloc(sizeof(MC));
515   mc->insn = malloc(sizeof(INSN_BUF));
516   if(strcmp(tac->proc.name->lexeme,"main")==0){
517     sprintf(mc->insn,"_%s:",tac->proc.name->lexeme);
518   }
519   else{
520     sprintf(mc->insn,"%s:",tac->proc.name->lexeme);
521   }
522   return mc;
```

```
523 }
524
525 MC* load_args(TAC* tac, FRME* e){
526   MC *mc = malloc(sizeof(MC));
527   mc->insn = malloc(sizeof(INSN_BUF));
528   MC* first = mc;
529   TOKENLIST* vars = tac->proc.args;
530   TOKEN* t;
531   int i = 0;
532   while(i < tac->proc.arity && vars != NULL){
533     mc->next = malloc(sizeof(MC));
534     mc->next->insn = malloc(sizeof(INSN_BUF));
535     t = lookup_loc(vars->name,e);
536     sprintf(mc->next->insn,"move $%s $a%d",t->lexeme,i);
537     mc = find_lst(mc);
538     vars = vars->next;
539     i++;
540   }
541   return first;
542 }
543
544 MC* new_cll(TAC* tac, FRME* e, AR* ar){
545   MC* mc = malloc(sizeof(MC));
546   mc->insn = malloc(sizeof(INSN_BUF));
547   MC* last = find_lst(mc);
548   int i=0;
549   TOKENLIST* args = tac->call.args;
550   while(i< tac->call.arity && args != NULL){
551     TOKEN* t = new_token(IDENTIFIER);
552     t->lexeme = (char*)calloc(1,2);
553     sprintf(t->lexeme,"a%d",i);
554     if(tac->call.args->name->type == IDENTIFIER){
555       last->next = new_smpl_ld(e,lookup_loc(args->name,e),t);
556     }
557     else{
558        last->next = new_smpl_ld(e,args->name,t);
559     }
560     last = find_lst(last);
561     args = args->next;
562     i++;
563   }
564   last = find_lst(last);
565   last->next = malloc(sizeof(MC));
566   last->next->insn = malloc(sizeof(INSN_BUF));
567   sprintf(last->next->insn,"jal %s",tac->call.name->lexeme);
568   return mc;
569 }
570
571 MC* gen_mc0(TAC* i, FRME* e, AR* curr){
572   MC* mc, *last;
573   CLSURE* f;
574   TOKEN* name;
575   if (i==NULL || i->op == tac_endproc)
576   {delete_constants(e); mc = new_func_rtn(i); return mc;}
577
578   switch (i->op) {
579   default:
580     printf("unknown type code %d (%p) in mmc_mcg\n",i->op,i);
581
582     case tac_plus:
```

```
583       mc = new_plus(i);
584       mc->next = gen_mc0(i->next,e,curr);
585       return mc;
586     case tac_minus:
587       mc = new_minus(i);
588       mc->next = gen_mc0(i->next,e,curr);
589       return mc;
590     case tac_div:
591       mc = new_div(i);
592       last = find_lst(mc);
593       last->next = gen_mc0(i->next,e,curr);
594       return mc;
595     case tac_mod:
596       mc = new_mod(i);
597       last = find_lst(mc);
598       last->next = gen_mc0(i->next,e,curr);
599       return mc;
600     case tac_mult:
601       mc = new_mult(i);
602       last = find_lst(mc);
603       last->next = gen_mc0(i->next,e,curr);
604       return mc;
605     case tac_innerproc:
606       name = i->proc.name;
607       f = find_fnc(name,e);
608       if (f == NULL){
609         f = malloc(sizeof(CLSURE));
610         f->env = e;
611         f->code = i;
612         declare_fnc(i->proc.name,f,e);
613         i = find_endproc(i->next);
614         mc = gen_mc0(i->next,e,curr);
615         return mc;
616       }
617     case tac_proc:
618       curr = calculate_frame(curr,i);
619       e->size = curr->size;
620       mc = new_prc(i,e);
621       last = find_lst(mc);
622       last->next = gen_frame(curr);
623       last = find_lst(last);
624       last->next = load_args(i,e);
625       last = find_lst(last);
626       last->next = gen_mc0(i->next,e,curr);
627       return mc;
628     case tac_load:
629       mc = new_ld(e,i,curr);
630       mc->next = gen_mc0(i->next,e,curr);
631       return mc;
632     case tac_store:
633       mc = new_str(i,e);
634       last = find_lst(mc);
635       last->next = gen_mc0(i->next,e,curr);
636       return mc;
637     case tac_if:
638       mc = new_ift(i,e);
639       last = find_lst(mc);
640       last->next = gen_mc0(i->next,e,curr);
641       return mc;
642     case tac_lbl:
```

```
643       mc = new_lbli(i);
644       last = find_lst(mc);
645       last->next = gen_mc0(i->next,e,curr);
646       return mc;
647     case tac_goto:
648       mc = new_gtl(i);
649       last = find_lst(mc);
650       last->next = gen_mc0(i->next,e,curr);
651       return mc;
652     case tac_call:
653       mc = save_frame(curr,e);
654       last = find_lst(mc);
655       last->next = new_cll(i,e,curr);
656       last = find_lst(last);
657       last->next = restore_frame(curr,e);
658       last = find_lst(last);
659       last->next = gen_mc0(i->next,e,curr);
660       last = find_lst(last);
661       last->next = call_func(i->call.name,i,e,curr);
662
663       return mc;
664     case tac_rtn:
665       mc = new_rtn(i,e,curr);
666       last = find_lst(mc);
667       last->next = gen_mc0(i->next,e,curr);
668       return mc;
669   }
670 }
671
672 MC* print_result() {
673
674     //print integer result
675     MC *mc = malloc(sizeof(MC));
676     mc->insn = malloc(sizeof(INSN_BUF));
677     mc->insn = "#print integer result";
678
679     MC* last = find_lst(mc);
680     last->next = malloc(sizeof(MC));
681     last->next->insn = malloc(sizeof(INSN_BUF));
682     last->next->insn = "move $a0 $v1";
683     last = find_lst(last);
684
685     last->next = make_syscall(PRINT_INT);
686
687     //print newline
688     last = find_lst(last);
689     last->next = malloc(sizeof(MC));
690     last->next->insn = malloc(sizeof(INSN_BUF));
691     last->next->insn = "li $a0 10";
692
693     last = find_lst(last);
694     last->next = make_syscall(PRINT_CHAR);
695
696     //exit
697     last = find_lst(last);
698     last->next = make_syscall(EXIT);
699     return mc;
700 }
701
702
```

```
703 MC* gen_mc(TAC* tac){
704   FRME* e = malloc(sizeof(FRME));
705   AR* global = malloc(sizeof(AR));
706   MC* mc = init_mc();
707   MC* last = find_lst(mc);
708   last->next = gen_globframe(tac,e,global);
709   last = find_lst(last);
710   FRME *ef = e;
711     while(ef != NULL){
712         BNDING* bindings = e->bindings;
713         while (bindings != NULL){
714             if(strcmp(bindings->name->lexeme,"main")==0){
715                 last->next = malloc(sizeof(MC));
716                 last->next->insn = malloc(sizeof(INSN_BUF));
717                 last->next->insn = "jal _main";
718                 last = find_lst(last);
719                 last->next = print_result();
720                 last = find_lst(last);
721                 last->next = call_func(bindings->name,NULL,e,global);
722                 return mc;
723             }
724             bindings = bindings->next;
725         }
726         ef = e->next;
727     }
728 }
```

```c
#include "environment.h"
#include "interpreter.h"
#include "C.tab.h"

extern VALUE* make_value_int(int);

VALUE *lookup_name(TOKEN * x, FRAME * frame){
    while(frame != NULL){
        BINDING *bindings = frame->bindings;
        while(bindings != NULL){
            if(bindings->name == x){
                return bindings->value;
            }
            bindings = bindings->next;
        }
        frame = frame->next;
    }
    return NULL;
}

VALUE *lookup_name_curr_frame(TOKEN * x, FRAME * frame){
    while(frame != NULL){
        BINDING *bindings = frame->bindings;
        while(bindings != NULL){
            if(bindings->name == x){
                return bindings->value;
            }
            bindings = bindings->next;
        }
        return NULL;
    }
}

VALUE *assign_to_name(TOKEN * x, FRAME * frame, VALUE * val){
    while(frame != NULL){
        BINDING *bindings = frame->bindings;
        while(bindings != NULL){
            if(bindings->name == x){
                bindings->value = val;
                return val;
            }
            bindings = bindings->next;
        }
        frame = frame->next;
    }
    printf("fatal: unbound variable!\n");exit(1);
}

VALUE *declare_name(TOKEN * x, FRAME * frame){
    BINDING *bindings = frame->bindings;
    BINDING *new = malloc(sizeof(BINDING));
    if(new != NULL){
        new->name = x;
        new->value = make_value_int(0);
        new->next = bindings;
        frame->bindings=new;
        return new->value;
    }
    printf("fatal: binding creation failed!\n");
}
```

```
61 VALUE *declare_func(TOKEN * x, VALUE* val, FRAME * frame){
62     BINDING *bindings = frame->bindings;
63     BINDING *new = malloc(sizeof(BINDING));
64     if(new != NULL){
65         new->name = x;
66         new->value = val;
67         new->next = bindings;
68         frame->bindings=new;
69         return new->value;
70     }
71     printf("fatal: binding creation failed!\n");
72 }
73
74
```