```
#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 }
```