```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 }
```