```
////////////////////////////////////////////////////////////////////////////
////Class:          CS 445
////Semester:       Fall 2011
////Assignment:     Homework 4
////Author:         Colby Blair
////File name:      tree.c
////////////////////////////////////////////////////////////////////////////

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdarg.h>

#include "tree.h"
//#include "symtabc.h" //comes with tree.h now
#include "tree_symtab_insert.h"
#include "tree_symtab_gen.h"
#include "parser.tab.h"
#include "main.h"

//yacc / bison stuff for parsing files from import statements
extern struct tree *YY_TREE;
extern char *YY_FNAME;
extern int lineno;
extern int colno;
extern FILE *yyin;

/*
This is the structure I think I am creating

        +---------------+
        |tree*          |
        |               |
        |prodrule=1     |
        |nkids=1        |
        |kids=          |---------------------- ...more
        |leaf=NULL      |       |
        +---------------+       |
                                |
                        +---------------+
                        |tree*          |
                        |               |
                        |prodrule=2     |
                        |nkids=0        |
                        |kids=NULL      |
                        |leaf=          |---------------
                        +---------------+               |
                                                +---------------+
                                                |tree_token*    |
                                                |               |
                                                |cat=256        |
                                                |text="int"     |
                                                |lineno=3       |
                                                |fname="main.as"|
                                                +---------------+
*/


struct tree *YY_TREE;


void tree_print_symtab_entry(void *p)
{
        if(p == NULL)
        {
                return;
        }

}


int tree_init()
{
```

```
        //init the global tree
        YY_TREE = NULL;

        //init the global symbol table
        SymTab_init(tree_print_symtab_entry);
}


int tree_del(struct tree *t)
{
        if(t == NULL)
        {
                return(1);
        }

        //go to children first
        int i;
        for(i=0; i < t->nkids; i++)
        {
                tree_del(t->kids[i]);
        }

        free(t->leaf);
        free(t);
        t == NULL;

        return(0);
}


int treeprint(struct tree *t, int depth)
{
        if(t == NULL)
        {
                return(1);
        }

        //printf("%*s %s: %d\n", depth*2, " ",
        //humanreadable(t->prodrule), t->nkids);

        //terminal
        if(t->leaf != NULL)
        {
                printf("%d %*s = \"%s\": %d\n",
                                        depth,
                                        depth + strlen(t->prodrule),
                                        t->leaf->text,
                                        t->prodrule, t->nkids);


        }
        else
        {
                printf("%d %*s: %d\n",  depth, depth + strlen(t->prodrule),
                                t->prodrule, t->nkids);
        }

        int i;
        for(i=0; i < t->nkids; i++)
        {
                treeprint(t->kids[i], depth+1);
        }

        //TODO: meaningful retval
        return(0);
}


//needs lots of trees as args
struct tree *tree_create_node(char *prodrule, int n_args, ...)
{
        //create the new node
        struct tree *retval = (struct tree*)malloc(sizeof(struct tree));
        //TODO: malloc check
```

```
        //init default member values
        retval->prodrule = strdup(prodrule);
        retval->nkids = n_args;
        retval->leaf = NULL;

        //init kids
        // start vargs
        register int i;
        va_list ap;
        // init kids
        int j;
        for(j = 0; j < MAX_KIDS; j++) {
                retval->kids[j] = NULL;
        }
        va_start(ap, n_args);

        // add kids
        // TODO: breaks if n_args bigger than MAX_KIDS
        for(i = 1; i <= n_args; i++) {
                retval->kids[i - 1] = va_arg(ap, struct tree*);
        }

        va_end(ap);

        return(retval);
}


//takes token attributes from lexer, and returns a tree node
struct tree *tree_create_node_from_token(int cat, char *text, int lineno,
                                                char *fname)
{
        //new tree token
        struct tree_token *new_token = (struct tree_token*)malloc(\
                                                sizeof(struct tree_token));
        new_token->cat = cat;
        new_token->text = strdup(text);
        new_token->lineno = lineno;
        new_token->fname = strdup(fname);

        //new (sub) tree
        struct tree *retval = (struct tree*)malloc(sizeof(struct tree));
        retval->prodrule = strdup(text);
        //retval->prodrule[strlen(retval->prodrule) - 1] = 0; //terminate
        retval->nkids = 0;
        retval->leaf = new_token;

        // init kids
        int j;
        for(j = 0; j < MAX_KIDS; j++) {
                retval->kids[j] = NULL;
        }

        return(retval);
}


//generate three address code (TAC)
// so far only traverses the tree and:
// 1. pushes and pops symbols in a symbol table stack according to scope
int tree_gen_tac(struct tree *t)
{
        if(t == NULL)
        {
                return(1);
        }

        //pre code generation ops; bring in imports, etc
        tree_preprocess(&t, 0);

        //generate code
        tree_process(t, 0);
```

```
        //post code generation ops; optimization, etc.
}


int tree_preprocess(struct tree **t, int depth)
{
        if((*t) == NULL)
        {
                return(1);
        }

        //preprocess ops
        if(strcmp((*t)->prodrule, "importDefinition") == 0)
        {
                tree_insert_importDefinition(&(*t));
        }

        int i;
        for(i=0; i < (*t)->nkids; i++)
        {
                tree_preprocess(&((*t)->kids[i]), depth+1);
        }

        return(0); //success
}

int tree_process(struct tree *t, int depth)
{
        if(t == NULL)
        {
                return(1);
        }

        //process ops
        // update symbol table
        tree_update_sym_tab(t);

        // assignmentExpression
        if(strcmp(t->prodrule, "assignmentExpression") == 0)
        {
                tree_gen_assignmentExpression(t);
        }

        int i;
        for(i=0; i < t->nkids; i++)
        {
                tree_process(t->kids[i], depth+1);
        }

        return(0); //success
}


void tree_get_subtree(char *prodrule, struct tree *from, struct tree **retval)
{
        if(from == NULL || prodrule == NULL || from->prodrule == NULL)
        {
                (*retval) = NULL;
                return;
        }

        //match
        if(strcmp(prodrule, from->prodrule) == 0)
        {
                (*retval) = from;
                return;
        }

        int i;
        for(i = 0; i < from->nkids; i++)
        {
                tree_get_subtree(prodrule, from->kids[i], &(*retval));
```

```
                if((*retval) != NULL)
                {
                        return;
                }
        }

        //didn't find anything down this path
        (*retval) = NULL;
        return;
}


//get an optional type that is somewhere within a subtree, usually:
// optionalTypeExpression
// |
// |
// ...
//  -ident
//      -leaf = "type" string
char *tree_get_opt_type(struct tree* t)
{
        char *type = NULL;

        if(t == NULL)
        {
                return(type); //NULL
        }

        //get larger type expression subtree
        struct tree *exp_subtree = NULL;
        tree_get_subtree("optionalTypeExpression", t, &exp_subtree);

        if(exp_subtree != NULL)
        {
                //get type subtree, which is usualy ident->Type
                struct tree *type_subtree = NULL;
                tree_get_subtree("ident", exp_subtree, &type_subtree);

                if(type_subtree != NULL
                        && type_subtree->nkids > 0
                        && type_subtree->kids[0] != NULL)
                {
                        type = type_subtree->kids[0]->prodrule;
                }
        }

        return(type);
}


char *tree_get_opt_aux_flag(struct tree* t)
{
        char *aux_flag = NULL;

        if(t == NULL)
        {
                return(aux_flag); //NULL
        }

        //get larger type expression subtree
        struct tree *exp_subtree = NULL;
        //room to be a parent of 'const' if we want more different aux flags
        tree_get_subtree("const", t, &exp_subtree);

        if(exp_subtree == NULL)
        {
                return(aux_flag); //NULL
        }
        else
        {
                // 'const'
                aux_flag = exp_subtree->prodrule;
```

```
                //add additional searches here for more aux flags
        }

        return(aux_flag);
}


//gets the ident string value, ie a variable name
char *tree_get_ident(struct tree* t)
{
        char *ident = NULL;

        if(t == NULL)
        {
                return(ident); //NULL
        }

        //get ident expression subtree
        struct tree *exp_subtree = NULL;
        tree_get_subtree("ident", t, &exp_subtree);

        if(exp_subtree == NULL || exp_subtree->kids[0] == NULL)
        {
                return(ident); //NULL
        }
        else
        {
                ident = exp_subtree->kids[0]->prodrule;
                //add additional searches here for more aux flags
        }

        return(ident);
}


int tree_update_sym_tab(struct tree *t)
{
        //terminals
        if(t->leaf != NULL)
        {
                //leave a scope
                if(t->leaf->cat == RCURLY)
                {
                        #ifdef DEBUG_SYMTAB
                        printf("Leaving scope. ");
                        printf(" Old symtab:\n");
                        SymTab_print(); //current symbol table
                        #endif
                        SymTab_leave_scope(); //the current scope in SymTab

                        #ifdef DEBUG_SYMTAB
                        printf(" New symtab:\n");
                        SymTab_print(); //current symbol table
                        #endif
                }
        }
        //nonterminals
        else
        {
                //method / function
                // 1. update symbol table
                // 2. enter a method / function scope
                if(strcmp(t->prodrule, "methodDefinition") == 0)
                {
                        //1. update symbol table
                        tree_symtab_insert_methodDefinition(t);

                        //2. enter a method / function scope
                        //SymTab_enter_scope(t->prodrule);
                        struct tree *sub_t = NULL;
                        char *prodrule = "ident";
                        tree_get_subtree(prodrule, t, &sub_t);
                        if(sub_t != NULL && sub_t->kids[0] != NULL
```

```
                                && sub_t->kids[0]->leaf != NULL)
                        {
                                char *scope_name = sub_t->kids[0]->leaf->text;
                                SymTab_enter_scope(scope_name);

                                #ifdef DEBUG_SYMTAB
                                printf("Entering scope '%s'. New symtab:\n",
                                                scope_name);
                                SymTab_print(); //current symbol table
                                #endif
                        }
                }
                //enter a class or package scope
                else if(
                        strcmp(t->prodrule, "classDefinition") == 0
                        || strcmp(t->prodrule, "packageDecl") == 0)
                {
                        //symbol table scope
                        //SymTab_enter_scope(t->prodrule);
                        struct tree *sub_t = NULL;
                        char *prodrule = "ident";
                        tree_get_subtree(prodrule, t, &sub_t);
                        if(sub_t != NULL && sub_t->kids[0] != NULL
                                && sub_t->kids[0]->leaf != NULL)
                        {
                                char *scope_name = sub_t->kids[0]->leaf->text;
                                SymTab_enter_scope(scope_name);

                                #ifdef DEBUG_SYMTAB
                                printf("Entering scope '%s'. New symtab:\n",
                                                scope_name);
                                SymTab_print(); //current symbol table
                                #endif
                        }
                }
                //function variable
                else if(strcmp(t->prodrule, "declarationStatement") == 0)
                {
                        tree_symtab_insert_variableDefinition(t);
                }
                //method / function paramater
                else if(strcmp(t->prodrule, "parameterDeclaration") == 0)
                {
                        tree_symtab_insert_parameterDeclaration(t);
                }
                //variableDefinition - class member
                else if(strcmp(t->prodrule, "variableDefinition") == 0)
                {
                        tree_symtab_insert_variableDefinition(t);
                }

        }
}


int tree_insert_importDefinition(struct tree **t)
{
        //t = importDefinition, so get the ident subtree
        struct tree *t_sub = NULL;
        tree_get_subtree("ident", (*t), &t_sub);
        if(t_sub == NULL)
        {
                return(1); //failure
        }

        //Get symbol pointer to variable name
        //TODO: need to use more generic getter
        char * import_name = t_sub->kids[0]->leaf->text;

        //Parse the import file
        tree_init(); //inits YY_TREE

        //open the file and store its reference in global variable yyin
```

```
        tree_import_ident_to_path(import_name, &YY_FNAME);

        //yyrestart for multiple file parsing
        FILE *yyfile = fopen(YY_FNAME, "r");
        yyrestart(yyfile); lineno = 1; colno = 1;
        //instead of - yyin = fopen(YY_FNAME,"r");

        if (yyin == NULL)
        {
                fprintf(stderr, "ERROR: import: Cannot open '%s'. ",\
                                                                YY_FNAME);
                #ifdef DEBUG_TREE
                fprintf(stderr, "Continuing anyway...\n");
                #else
                fprintf(stderr, "Cannot continue.\n");
                exit(ERROR_SEMANTIC);
                #endif
        }
        else
        {
                //print the name of the file
                printf("%s\n", YY_FNAME);

                //parse import file
                yyparse();

                //delete the importDefinition subtree...
                tree_del((*t));

                //...and replace it with the newly parsed YY_TREE
                //TODO: do we really want to replace with a as3CompilationUnit
                // subtree, or it's children? Leaving for now.
                (*t) = YY_TREE;
        }

        return(0); //success
}


void tree_import_ident_to_path(char *fname, char **retval)
{
        if(fname == NULL)
        {
                return(NULL);
        }

        //replace '.' chars with dir delims. Sorry windows, only linux paths
        // for now
        int i;
        for(i = 0; i < strlen(fname); i++)
        {
                if(fname[i] == '.')
                {
                        fname[i] = '/';
                }
        }

        //append an ".as" to the end
        char *postfix = ".as";
        (*retval) = (char*)malloc( (strlen(fname) + strlen(postfix)) \
                                        * sizeof(char));
        sprintf( (*retval), "%s%s", fname, postfix);

}
```