```
/* chad c d clark   < clarkch @ cpsc . ucalgary . ca >
 *
 * cpsc 411     lec ??
 * winter 2002  lab 02
 *
 * assignment #1 - a first stab.
 *
 * file: as1tree.c
 * purpose: contains the functions that operate on the tree structures to be
 *      used in the syntax tree.
 *
 *
 */


#include "as1tree.h"    /* holds the struct stree_node */
#include <stdlib.h>     /* for malloc() */
#include <stdio.h>      /* for printf(), stderr */


/* we use this to number the labels generated in our target code. */
extern int label_counter;



/* functions for stmt -> rules ************************/

/* makeIFNODE() builds a tree segment for an IF statement of the form:
 *      IF expr THEN true_stmt ELSE false_stmt
 *
 * the final stucture is:
 *
 *      IF
 *      |
 *      expr - true_stmt - false_stmt
 *
 */
struct stree_node * makeIFnode(struct stree_node *if_expr,
                struct stree_node *true_stmt,
                struct stree_node *false_stmt) {

        struct stree_node *new;

        new = (struct stree_node *) malloc(sizeof(struct stree_node));

        new->type = IF_NODE;
        new->numval = 0;
        new->idval = NULL;
        new->sibling = NULL;

        new->child = if_expr;
        if_expr->sibling = true_stmt;
        true_stmt->sibling = false_stmt;

        return(new);

}       /* makeIFnode() */


/* makeWHILEnode() builds a tree segement for a WHILE statment of the form:
 *      WHILE expr DO stmt
 *
 * the final stucture looks like:
 *
```

```
 *      WHILE
 *        |
 *      expr - stmt
 *
 */
struct stree_node * makeWHILEnode(struct stree_node *while_expr,
                struct stree_node *do_stmt) {

        struct stree_node *new;

        new = (struct stree_node *) malloc(sizeof(struct stree_node));

        new->type = WHILE_NODE;
        new->numval = 0;
        new->idval = NULL;
        new->sibling = NULL;

        new->child = while_expr;
        while_expr->sibling = do_stmt;

        return(new);

}       /* makeWHILEnode */


/* makeDOnode() builds a tree segment for a DO statement of the form:
 *      DO stmt UNTIL expr
 *
 * the final structure looks like:
 *
 *      DO
 *      |
 *      stmt - expr
 *
 */
struct stree_node * makeDOnode(struct stree_node *do_stmt,
                struct stree_node *while_expr) {

        struct stree_node *new;

        new = (struct stree_node *) malloc(sizeof(struct stree_node));

        new->type = DO_NODE;
        new->numval = 0;
        new->idval = NULL;
        new->sibling = NULL;

        new->child = do_stmt;
        do_stmt->sibling = while_expr;

        return(new);


}       /* makeDOnode */


/* makeASSIGNnode() builds a tree segment for an ASSING statement of the form:
 *      ID ASSIGN expr
 *
 * the final stucture build looks like:
 *
 *      ASSIGN
 *         |
 *        ID   -  expr
 *
```

```
 */
struct stree_node * makeASSIGNnode(struct stree_node *id_node,
                struct stree_node *an_expr) {

        struct stree_node *new;

        new = (struct stree_node *) malloc(sizeof(struct stree_node));

        new->type = ASSIGN_NODE;
        new->numval = 0;
        new->idval = NULL;
        new->sibling = NULL;

        new->child = id_node;
        id_node->sibling = an_expr;

        return(new);

}       /* makeASSIGNnode() */


/* makeREADnode() builds a tree segment for a READ statement of the form:
 *      READ ID
 *
 * the structure build looks like:
 *
 *      READ
 *       |
 *       ID
 *
 */
struct stree_node *makeREADnode(struct stree_node *id_node) {

        struct stree_node *read;

        read = (struct stree_node *) malloc(sizeof(struct stree_node));

        read->type = READ_NODE;
        read->numval = 0;
        read->idval = NULL;
        read->sibling = NULL;

        read->child = id_node;

        return(read);

}       /* makeREADnode() */


/* makePRINTnode() builds a tree segment for a statement of the form:
 *      PRINT expr
 *
 *      the returned segment has the structure:
 *
 *      PRINT
 *        |
 *      expr
 *
 */
struct stree_node *makePRINTnode(struct stree_node *an_expr) {

        struct stree_node *print;

        print = (struct stree_node *) malloc(sizeof(struct stree_node));
```

```
        print->type = PRINT_NODE;
        print->numval = 0;
        print->idval = 0;
        print->sibling = NULL;

        print->child = an_expr;

        return(print);

}       /* makePRINTnode() */


/* makeBEGINnode() builds a tree segemnt for a statement of the form:
 *      BEGIN stmtlist END
 *
 * the built and returned segement has the sturcture:
 *
 *      BEGIN
 *        |
 *      stmlist
 *
 */
struct stree_node *makeBEGINnode(struct stree_node *slist) {

        struct stree_node *new;

        new = (struct stree_node *) malloc(sizeof(struct stree_node));

        new->type = BEGIN_NODE;
        new->numval = 0;
        new->idval = NULL;
        new->sibling = NULL;

        new->child = slist;

        return(new);

}       /* makeBEGINnode */


/* functions for stmtlist -> rules ***************************/

/* makeSTMTLISTnode() builds a tree segment for a statemnt of the form:
 *      stmt stmtlist
 *
 * the returned structure looks like:
 *
 *      STMTLIST
 *         |
 *        stmt  -  stmtlist
 *
 */
struct stree_node *makeSTMTLISTnode(struct stree_node *a_stmt,
                struct stree_node *a_stmtlist) {

        struct stree_node *new;

        new = (struct stree_node *) malloc(sizeof(struct stree_node));

        new->type = STMTLIST_NODE;
        new->numval = 0;
        new->idval = NULL;
        new->sibling = NULL;

        new->child = a_stmt;
```

```
        a_stmt->sibling = a_stmtlist;

        return(new);
}



/* functions for expr-> rules *************************/

/* makeADDnode() builds a tree segment for a statement of the form:
 *      term ADD right_term
 *
 * the segment has the form
 *
 *      ADD
 *       |
 *      term - right_term
 *
 */
struct stree_node *makeADDnode(struct stree_node *a_term,
                struct stree_node * right_term) {

        struct stree_node *add;

        add = (struct stree_node *) malloc(sizeof(struct stree_node));

        add->type = ADD_NODE;
        add->numval = 0;
        add->idval = NULL;
        add->sibling = NULL;

        add->child = a_term;
        a_term->sibling = right_term;

        return(add);

}       /* makeADDnode() */


/* makeSUBnode() builds a tree segment for a statement of the form:
 *      term SUB right_term
 *
 * the segment has the form
 *
 *      SUB
 *       |
 *      term - right_term
 *
 */
struct stree_node *makeSUBnode(struct stree_node *a_term,
                struct stree_node * right_term) {

        struct stree_node *sub;

        sub = (struct stree_node *) malloc(sizeof(struct stree_node));

        sub->type = SUB_NODE;
        sub->numval = 0;
        sub->idval = NULL;
        sub->sibling = NULL;

        sub->child = a_term;
        a_term->sibling = right_term;

        return(sub);
```

```
}       /* makeSUBnode() */



/* functions for term -> rules *************************/


/* makeMULnode() builds a tree segment for a statement of the form:
 *      factor MUL right_factor
 *
 * the segment has the form
 *
 *      MUL
 *       |
 *      factor - right_factor
 *
 */
struct stree_node *makeMULnode(struct stree_node *a_factor,
                struct stree_node * right_factor) {

        struct stree_node *mul;

        mul = (struct stree_node *) malloc(sizeof(struct stree_node));

        mul->type = MUL_NODE;
        mul->numval = 0;
        mul->idval = NULL;
        mul->sibling = NULL;

        mul->child = a_factor;
        a_factor->sibling = right_factor;

        return(mul);

}       /* makeMULnode() */


/* makeDIVnode() builds a tree segment for a statement of the form:
 *      factor DIV right_factor
 *
 * the segment has the form
 *
 *      DIV
 *       |
 *      factor - right_factor
 *
 */
struct stree_node *makeDIVnode(struct stree_node *a_factor,
                struct stree_node * right_factor) {

        struct stree_node *div;

        div = (struct stree_node *) malloc(sizeof(struct stree_node));

        div->type = DIV_NODE;
        div->numval = 0;
        div->idval = NULL;
        div->sibling = NULL;

        div->child = a_factor;
        a_factor->sibling = right_factor;

        return(div);
```

```
}       /* makeDIVnode() */


/* functions for factor -> rules *************************/

/* makeIDnode() returns a node such that an identifier name is stored
 * in the node's 'char *idval' member.
 *
 */
struct stree_node *makeIDnode(char *ident) {

        struct stree_node *new;

        new = (struct stree_node *) malloc(sizeof(struct stree_node));

        new->type = ID_NODE;
        new->numval = 0;
        new->sibling = NULL;
        new->child = NULL;

        new->idval = (char*) malloc(strlen(ident)+1);
        strcpy(new->idval, ident);

        return(new);
}

/* makeNUMnode() returns a node such that an integer (whose string value is
 * represented by 'char *num') is stored in the node's 'int numval' member.
 *
 * Negative numbers are allowed as atoi() is used. For more detailed info on
 * acceptable string formats refer to the man page for atoi (section 3).
 *
 */
struct stree_node *makeNUMnode(char *num) {

        struct stree_node *new;

        new = (struct stree_node *) malloc(sizeof(struct stree_node));

        new->type = NUM_NODE;
        new->idval = NULL;
        new->sibling = NULL;
        new->child = NULL;

        new->numval = atoi(num);

        return(new);
}


/* A function to print out the syntax tree for debugging.
 *
 * print_stree() is a recursive fuction that looks at the current node
 * type and prints out the node type.  the function then calls itself
 * for each of it's children.
 *
 * terminal nodes print out the value's they store.
 *
 * also each call is made with an increasing value that is used to
 * print out some preceding spaces to make the tree sort of readable.
 * a program with a very deep tree won't look too nice but this works
 * for testing.
 *
 */
void print_stree(struct stree_node *node, int spaces) {
```

```
        int i = 0;      /* for loop vbl */

        /* print some spaces to make our tree nice to look at and understand */
        for (i = 0; i < spaces; i++) {
                printf(" ");
        }

        /* the only time I've seen this NULL is at the end of a stmtlist.
         * the real solution would be to add an END_NODE but this is quicker
         * and also catches a rouge NULL pointer if it should pop up.
         *
         * This was FIXED - added a check to the STMTLIST case.  Now this check
         * should never find a NULL pointer. (So I think :)
         */
        if(!node) {

                fprintf(stderr, "Warning: NULL pointer in syntax tree.\n");
                return;
        }

        switch(node->type) {
        /* for info on what each node looks like refer to comments found
         * with the makeXXXXnode() functions in this file.
         */

                case(IF_NODE):
                        printf("IF_NODE\n");
                        print_stree(node->child, spaces+1);
                        print_stree(node->child->sibling, spaces+1);
                        print_stree(node->child->sibling->sibling, spaces+1);
                        break;

                case(WHILE_NODE):
                        printf("WHILE_NODE\n");
                        print_stree(node->child, spaces+1);
                        print_stree(node->child->sibling, spaces+1);
                        break;

                case(DO_NODE):
                        printf("DO_NODE\n");
                        print_stree(node->child, spaces+1);
                        print_stree(node->child->sibling, spaces+1);
                        break;

                case(ASSIGN_NODE):
                        printf("ASSIGN_NODE\n");
                        print_stree(node->child, spaces+1);
                        print_stree(node->child->sibling, spaces+1);
                        break;

                case(READ_NODE):
                        printf("READ_NODE\n");
                        print_stree(node->child, spaces+1);
                        break;

                case(PRINT_NODE):
                        printf("PRINT_NODE\n");
                        print_stree(node->child, spaces+1);
                        break;

                case(BEGIN_NODE):
                        printf("BEGIN_NODE\n");
                        print_stree(node->child, spaces+1);
                        break;
```

```
                case(STMTLIST_NODE):
                        printf("STMTLIST_NODE\n");
                        print_stree(node->child, spaces+1);
                        /* the end of a stmtlist is denoted by a NULL pointer */
                        if(node->child->sibling)
                                print_stree(node->child->sibling, spaces+1);
                        break;

                case(ADD_NODE):
                        printf("ADD_NODE\n");
                        print_stree(node->child, spaces+1);
                        print_stree(node->child->sibling, spaces+1);
                        break;

                case(SUB_NODE):
                        printf("SUB_NODE\n");
                        print_stree(node->child, spaces+1);
                        print_stree(node->child->sibling, spaces+1);
                        break;

                case(MUL_NODE):
                        printf("MUL_NODE\n");
                        print_stree(node->child, spaces+1);
                        print_stree(node->child->sibling, spaces+1);
                        break;

                case(DIV_NODE):
                        printf("DIV_NODE\n");
                        print_stree(node->child, spaces+1);
                        print_stree(node->child->sibling, spaces+1);
                        break;

                case(ID_NODE):
                        printf("ID_NODE idval: %s\n", node->idval);
                        break;

                case(NUM_NODE):
                        printf("NUM_NODE numval: %d\n", node->numval);
                        break;

                default:
                        printf("UNKNOWN NODE TYPE.  Sorry dude.   :(\n");

        }       /* switch */


}       /* print_stree() */


/* A debugging function used to search for a particular type of node in
 * the stree.
 *
 * args:
 *      the first argument is the node type (see as1tree.h)
 *      the second argument is the tree segment to be searched.
 *
 * returns:
 *      0 on failure.
 *      1 on success.
 *
 * also this fuction calls print_stree() when it finds a node of
 * the type searched for.
 *
 */
```

```
int find_node(int type, struct stree_node *node) {

        if (node->type == type) {
                print_stree(node, 0);
                return(1);
        }
        else {
        /* the && 's are used to short circuit the call to find_node() */

                if (node->sibling && find_node(type, node->sibling)) {
                        return(1);
                }
                if (node->child && find_node(type, node->child)) {
                        return(1);
                }
        }
        return(0);
}


/* function to recursivly delete a tree *********************
 *
 * delete_stree() simply recurses throught the tree and deletes leaf nodes.
 * before deleting a node we free 'char *idval' if need be.
 *
 */
void delete_stree(struct stree_node *node) {

        /* if we have dependants deal with them first */
        if(node->sibling) delete_stree(node->sibling);
        if(node->child) delete_stree(node->child);

        /* we should be able to nuke this node now that we've freed the
         * dependants.
         *
         * first we clean up any info it may contain though.
         */
        if(node->idval) free(node->idval);

        free(node);

}       /* delete_stree() */


/* function to generate the stack machine code *************
 *
 * gen_code() prints out the stack machine code to stdout that corrosponds
 * to the tree structure pointed to by the function's only argument.
 *
 * this function is quite simmilar to print_stree().  all it does is looks
 * at the node passed to it to determine what target code should be printed.
 *
 * when code needs to be defined by a sub-tree the code is filled in by a
 * call to code_gen() with a pointer to the subtree as an argument.
 *
 * this nifty recusion works because the results of sub-trees are put on to
 * the stack and that code gets executed before we get back to the node that
 * made the recursive call.  more importantly returnish like values get left
 * on the stack for the calling node to use after the sub-tree code is executed.
 *
 */
void gen_code(struct stree_node *node) {

        /* node pointers used to keep track of nodes durring code generation */
        struct stree_node *expr_node;
```

```
        struct stree_node *true_node;
        struct stree_node *false_node;
        struct stree_node *do_node;
        struct stree_node *until_expr;
        struct stree_node *id_node;
        struct stree_node *val_node;
        struct stree_node *a_node;
        struct stree_node *a_term;
        struct stree_node *r_term;
        struct stree_node *a_factor;
        struct stree_node *r_factor;

        /* used to store variable label values durring generating of code with
         * loops in it.
         */
        int false_label;
        int end_label;
        int start_label;

        /* don't do much with NULL pointers */
        if(!node) {

                fprintf(stderr, "Warning: NULL pointer in syntax tree.\n");
                return;
        }

        switch(node->type) {

                case(IF_NODE):
                /* Okay here is the plan:
                 * - evaluate the expresion
                 * - if false jump to false_stmt code
                 * - else fall through to true_stmt code
                 * - after the true_stmt code jump over the false_stmt code
                 *   to the end_label
                 * - after the false_stmt code continue on to the end_label
                 *
                 */
                        expr_node = node->child;
                        true_node = expr_node->sibling;
                        false_node = true_node->sibling;

                        false_label = label_counter++;
                        end_label = label_counter++;

                        gen_code(expr_node);
                        printf("cJUMP L%d\n", false_label);

                        gen_code(true_node);
                        printf("JUMP L%d\n", end_label);

                        printf("L%d:\n", false_label);
                        gen_code(false_node);

                        printf("L%d:\n", end_label);
                        break;

                case(WHILE_NODE):
                /* The plan is:
                 * - mark the begining of the expr (start_label)
                 * - evaluate the expr code
                 * - if the expression is 0 jump to the end_label
                 * - else fall through to the do_stmt code
                 * - generate the do_stmt code
                 * - jump back to the begining of expr (start_label)
```

```
 *   (to evalate and test it again)
 * - mark the end of the loop (end_label) so we can
 *   continue on when expr evaluates to zero
 *
 */
        expr_node = node->child;
        do_node = expr_node->sibling;

        start_label = label_counter++;
        end_label = label_counter++;

        printf("L%d:\n", start_label);
        gen_code(expr_node);
        printf("cJUMP L%d\n", end_label);

        gen_code(do_node);
        printf("JUMP L%d\n", start_label);

        printf("L%d:\n", end_label);
        break;

case(DO_NODE):
/* The plan is:
 * - mark the begining of the loop (start_label)
 * - generate the loop code (do_node)
 * - evaluate the condition (until_expr)
 * - if false jump to the end of do statement
 * - condition is not zero so jump back to the
 *   beginging (start_label)
 * - mark the end of the do statment (end_label) so we
 *   can continue on when the condition is zero
 *
 */
/* WHOOPS THIS IS A DO-WHILE LOOP, WE WANT A DO-UNTIL LOOP !!
*       do_node = node->child;
*       until_expr = do_node->sibling;
*
*       start_label = label_counter++;
*       end_label = label_counter++;
*
*       printf("L%d:\n", start_label);
*       gen_code(do_node);
*
*       gen_code(until_expr);
*       printf("cJUMP L%d\n", end_label);
*
*       printf("JUMP L%d\n", start_label);
*
*       printf("L%d:\n", end_label);
*       break;
*/
/* The real plan is:
 * - mark the begining of the loop (start_label)
 * - generate the loop code (do_node)
 * - evaluate the expresion (until_expr)
 * - if false jump to the top of the loop
 * - otherwise fall through and continue on
 *
 */
        do_node = node->child;
        until_expr = do_node->sibling;

        start_label = label_counter++;

        printf("L%d:\n", start_label);
```

```
                gen_code(do_node);

                gen_code(until_expr);
                printf("cJUMP L%d\n", start_label);

                break;

        case(ASSIGN_NODE):
        /* Here we just:
         * - evaluate the expresion
         * - print code to load the variable (register) with
         *   the expresion's value.
         *
         */
                id_node = node->child;
                expr_node = id_node->sibling;

                gen_code(expr_node);
                printf("LOAD %s\n", id_node->idval);

                break;

        case(READ_NODE):
        /* This is an easy step, just print code to read into a
         * variable (register).
         *
         */
                id_node = node->child;
                printf("READ %s\n", id_node->idval);
                break;

        case(PRINT_NODE):
        /* generate the code to evaluate an expresion.
         * the expresion's value gets left on the stack so
         * just print code to print and pop it.
         *
         */
                val_node = node->child;
                gen_code(val_node);
                printf("PRINT\n");
                break;

        case(BEGIN_NODE):
        /* We don't need to worry about begin nodes too much.
         * They just refer to a stmtlist node so generate the
         * code for the statement list.
         *
         */
                gen_code(node->child);
                break;

        case(STMTLIST_NODE):
        /* print the code for the current (ie. this) statement
         * then print the code for the next statment in the list.
         *
         */
                a_node = node->child;
                while(a_node) {
                        gen_code(a_node);
                        a_node = a_node->sibling;
                }
                break;

        case(ADD_NODE):
        /* The order is important here.
```

```
         * - generate the code for the _right_hand_ term.  It's
         *   evaluated value will end up on the stack.
         * - generate the code for the _left_hand_ term.  It's
         *   evaluated value will end up on _top_ of the right
         *   hand term.
         * - print the code to perform the addition of the two
         *   terms.
         *
         */
                a_term = node->child;
                r_term = a_term->sibling;

                gen_code(r_term);
                gen_code(a_term);

                printf("OP2 +\n");
                break;

        case(SUB_NODE):
        /* The order is important here.
         * - generate the code for the _right_hand_ term.  It's
         *   evaluated value will end up on the stack.
         * - generate the code for the _left_hand_ term.  It's
         *   evaluated value will end up on _top_ of the right
         *   hand term.
         * - print the code to perform the subtraction of the two
         *   terms.  The right hand term will be subtracted from
         *   the left hand term.
         *
         */
                a_term = node->child;
                r_term = a_term->sibling;

                gen_code(r_term);
                gen_code(a_term);

                printf("OP2 -\n");
                break;

        case(MUL_NODE):
        /* The order is important here.
         * - generate the code for the _right_hand_ factor.  It's
         *   evaluated value will end up on the stack.
         * - generate the code for the _left_hand_ factor.  It's
         *   evaluated value will end up on _top_ of the right
         *   hand factor.
         * - print the code to perform the multiplication of the
         *   two factors.
         *
         */
                a_factor = node->child;
                r_factor = a_factor->sibling;

                gen_code(r_factor);
                gen_code(a_factor);

                printf("OP2 *\n");
                break;

        case(DIV_NODE):
        /* The order is important here.
         * - generate the code for the _right_hand_ factor.  It's
         *   evaluated value will end up on the stack.
         * - generate the code for the _left_hand_ factor.  It's
         *   evaluated value will end up on _top_ of the right
```

```
                 *   hand factor.
                 * - print the code to perform the division of the two
                 *   factors.  The right hand factor will be divided _into_
                 *   the left hand term. (ie. the left factor will be
                 *   divided by the right).
                 *
                 */
                        a_factor = node->child;
                        r_factor = a_factor->sibling;

                        gen_code(r_factor);
                        gen_code(a_factor);

                        printf("OP2 /\n");
                        break;

                case(ID_NODE):
                /* Here we print code to push the name of the variable
                 * (register) on to the stack.  This means that it's
                 * value is used and makes our code generation easier.
                 *
                 * eg: print x;    which looks like:    PRINT
                 *                                        |
                 *                                      ID
                 *    develops like:
                 *     - gen_code(PRINT_NODE) calls gen_code(ID_NODE)
                 *       - gen_code(ID_NODE) prints "PUSH x\n"
                 *     - gen_code(PRINT_NODE) prints "PRINT\n"
                 *
                 */
                        printf("rPUSH %s\n", node->idval);
                        break;

                case(NUM_NODE):
                /* This is quite simmilar to case(ID_NODE) just looked at.
                 * We just put the numeric value on the stack so it can be
                 * used as like any value.
                 *
                 */
                        printf("cPUSH %d\n", node->numval);
                        break;

                default:
                /* I don't think this this should come up but what the hey.
                 * It's better to get a message when it errors then to not.
                 */
                        fprintf(stderr, "Uh-oh!  Bad node type: %d\n",
                                        node->type);
                        break;

        }

}       /* gen_code() */
```