```
/* chad c d clark   < clarkch @ cpsc . ucalgary . ca >
 *
 * cpsc 411     lec ??
 * winter 2002  lab 02
 *
 * assignment #1 - a first stab.
 *
 * file: as1pars.c
 * purpose: a basic parser for minisculus.
 *
 *
 */


/* Set DEBUG to 1 for extra printf()'s
 * Set DEBUG to 0 for fewer printf()'s
 *
 */
#define DEBUG   0



/* ## Includes ################################################################ */

/* printf, etc */
#include <stdio.h>

/* #defines for values of the tokens. */
#include "as1tokens.h"

/* the tree stucture and functions */
#include "as1tree.h"



/* ## externals ################################################################ */

/* the text of the current token */
extern char * yytext;

/* gets the next token */
extern int yylex(void);

/* the current token (as a type)
 * see as1tokens.h for the definitions
 */
extern int curr_token;



/* ## Function Prototypes ###################################################### */

/* The prog() function performs the recursion for the BNF rule:
 *      prog -> stmt
 */
struct stree_node * prog();

/* The stmt() function performs the recursion for the BNF rule:
 *      stmt -> IF expr THEN stmt ELSE stmt
 *              | WHILE expr DO stmt
 *              | DO stmt UNTIL expr
 *              | READ ID
 *              | ID ASSIGN expr
 *              | PRINT expr
```

```
 *              | BEGIN stmtlist END
 */
struct stree_node * stmt();

/* The stmtlist() function performs the recursion for the BNF rule:
 *      stmtlist -> stmtlist stmt SEMICOLON
 *              |
 */
struct stree_node * stmtlist();

/* The expr() function performs the recursion for the BNF rule:
 *      expr -> expr addop term
 *              | term
 *
 * This is done via an equivalent EBNF rule:
 *      expr -> term { addop term }
 */
struct stree_node * expr();

/* The addop() function performs the recursion for the BNF rule:
 *      addop -> ADD
 *              | SUB
 *
 * This ammounts to just checking for syntax errors and eating up a token.
 */
struct stree_node * addop();

/* The term() function performs the recursion for the BNF rule:
 *      term -> term mulop factor
 *              | factor
 *
 * This is done via an equivalent EBNF rule:
 *      term -> factor { mulop factor }
 */
struct stree_node * term();

/* The mulop() function performs the recursion for the BNF rule:
 *      mulop -> MUL
 *              | DIV
 *
 * This ammounts to just checking for syntax errors and eating up a token.
 */
struct stree_node * mulop();

/* The factor() function performs the recursion for the BNF rule:
 *      factor -> LPAR expr RPAR
 *              | ID
 *              | NUM
 *              | SUB NUM
 */
struct stree_node * factor();



/* ## Functions ############################################################ */

/* parse_error() wines to stderr and calls the parse process quits.
 * this function gets called when the syntax read in seems to be wrong.
 *
 */
void parse_error() {

        fprintf(stderr, "\nPARSE ERROR: tough luck  :( ");
        fprintf(stderr, "\n\tHint: %s\n", yytext);
        exit(-1);
```

```
}


/* match_token() is an auxillary function that checks the current token
 * against the function's only argument.
 *
 * If they match the current token is advanced one token in the input.
 * If they don't match parse_error() is called.
 *
 */
void match_token(int token) {

        if (DEBUG) printf("match_token trying for %d\n",token);

        if (token == curr_token)
                curr_token = yylex();
        else
                parse_error();
}



/* prog() deals with the rule:
 *      prog -> stmt
 * by calling stmt() and returning the result of stmt() as the result of prog()
 *
 */
struct stree_node * prog() {

        struct stree_node *ret_tree;
        if (DEBUG) printf("prog(): token = %s\n", yytext);

        ret_tree = stmt();

        if (DEBUG) printf(" -> end of prog()\n");
        return(ret_tree);
}


/* stmt() deals with the rule:
 *      stmt -> IF expr THEN stmt ELSE stmt
 *              | WHILE expr DO stmt
 *              | DO stmt UNTIL expr
 *              | READ ID
 *              | ID ASSIGN expr
 *              | PRINT expr
 *              | BEGIN stmtlist END
 *
 * by examining the first token and then calling other rule's functions
 * depending on the first token.  Finally a node in the syntax tree is
 * made and returned as the recursion unwinds.
 *
 */
struct stree_node * stmt() {

        /* variables used for storage of arguments while building nodes. */
        char * ident;
        struct stree_node *ident_node;
        struct stree_node *an_expr;
        struct stree_node *if_expr;
        struct stree_node *then_stmt;
        struct stree_node *else_stmt;
        struct stree_node *do_stmt;
        struct stree_node *while_expr;
        struct stree_node *until_expr;
```

```
        struct stree_node *slist;
        struct stree_node *ret_tree;


        if (DEBUG) printf("stmt(): token = %s -> ", yytext);

        switch(curr_token) {


                case (T_IF):
                /* stmt -> IF expr THEN stmt ELSE stmt */

                        if (DEBUG) printf("IF type\n");
                        match_token(T_IF);
                        if (DEBUG) printf("IF\n");
                        if_expr = expr();
                        match_token(T_THEN);
                        if (DEBUG) printf("THEN\n");
                        then_stmt = stmt();
                        match_token(T_ELSE);
                        if (DEBUG) printf("ELSE\n");
                        else_stmt = stmt();

                        ret_tree = makeIFnode(if_expr, then_stmt, else_stmt);

                        if (DEBUG) printf(" -> end of stmt()\n");
                        return(ret_tree);
                        break;

                case (T_WHILE):
                /* stmt -> WHILE expr DO stmt */

                        if (DEBUG) printf("WHILE type\n");
                        match_token(T_WHILE);
                        while_expr = expr();
                        match_token(T_DO);
                        do_stmt = stmt();

                        ret_tree = makeWHILEnode(while_expr, do_stmt);

                        if (DEBUG) printf(" -> end of stmt()\n");
                        return(ret_tree);
                        break;

                case (T_DO):
                /* stmt -> DO stmt UNTIL expr */

                        if (DEBUG) printf("DO type\n");
                        match_token(T_DO);
                        do_stmt = stmt();
                        match_token(T_UNTIL);
                        until_expr = expr();

                        ret_tree = makeDOnode(do_stmt, until_expr);

                        if (DEBUG) printf(" -> end of stmt()\n");
                        return(ret_tree);
                        break;

                case (T_READ):
                /* stmt -> READ ID */

                        if (DEBUG) printf("READ type\n");
                        match_token(T_READ);
                        ident_node = makeIDnode(yytext);
```

```
                        match_token(T_ID);

                        ret_tree = makeREADnode(ident_node);

                        if (DEBUG) printf(" -> end of stmt()\n");
                        return(ret_tree);
                        break;

                case (T_ID):
                /* stmt -> ID ASSIGN expr */

                        if (DEBUG) printf("ID type\n");

                        ident_node = makeIDnode(yytext);

                        match_token(T_ID);
                        match_token(T_ASSIGN);
                        an_expr = expr();

                        ret_tree = makeASSIGNnode(ident_node, an_expr);

                        if (DEBUG) printf(" -> end of stmt()\n");
                        return(ret_tree);
                        break;

                case (T_PRINT):
                /* stmt -> PRINT expr */

                        if (DEBUG) printf("PRINT type\n");
                        match_token(T_PRINT);

                        return(makePRINTnode(expr()));

                        break;

                case (T_BEGIN):
                /* stmt -> BEGIN stmtlist END */

                        if (DEBUG) printf("BEGIN type\n");

                        match_token(T_BEGIN);
                        slist = stmtlist();
                        match_token(T_END);

                        ret_tree = makeBEGINnode(slist);

                        return(ret_tree);
                        break;

                default:
                /* we should never get here */

                        if (DEBUG) printf("not a valid statement type\n");
                        parse_error();
                        if (DEBUG) printf(" -> end of stmt()\n");
                        return((struct stree_node*)4);
                                /* 4 seems convenient (and used in lab) */
                                /* prob a bad pointer value though! */
                        break;

        }       /* switch */
}       /* stmt() */


/* stmtlist() deals with the rule:
```

```
 *      stmtlist -> stmtlist stmt SEMICOLON
 *              |
 *
 * by treating the rule as the rule:
 *      stmtlist -> stmt SEMICOLON stmtlist
 *              |
 *
 * epsilon (denoted by a NULL pointer) is generated when we get to an END
 * token as END is the only element of stmtlist's follow set.
 *
 */
struct stree_node * stmtlist() {

        struct stree_node *a_stmt;
        struct stree_node *rest_list;

        if (DEBUG) printf("stmtlist(): token = %s\n", yytext);
        if (curr_token != T_END) {

                a_stmt = stmt();
                match_token(T_SEMICOLON);
                rest_list = stmtlist();

                if (DEBUG) printf(" -> end of stmtlist()\n");

                return(makeSTMTLISTnode(a_stmt, rest_list));


        }       /* if */

        if (DEBUG) printf(" -> end of stmtlist()\n");
        return(0);  /* null pointer */

}       /* stmtlist() */



/* The expr() function performs the recursion for the BNF rule:
 *      expr -> expr addop term
 *              | term
 *
 * This is done via an equivalent EBNF rule:
 *      expr -> term { addop term }
 *
 *
 * thanks to K.C. Louden's _Compiler_Construction_ (pp 146) for showing the
 * BNF [ a -> a b c | c ] to be equivalent to the EBNF [ a -> c { b c } ].
 *
 *
 */
struct stree_node * expr() {

        /* pointers to nodes that get returned to us by other functions.
         * we use these to build our node.
         */
        struct stree_node *term_expr;
        struct stree_node *right;
        struct stree_node *node;

        if (DEBUG) printf("expr(): token = %s\n", yytext);

        term_expr = term();

        /* so long as we still have an addop keep chaining terms together */
```

```
        while (curr_token == T_ADD  ||  curr_token == T_SUB) {

                if (curr_token == T_ADD) {

                        match_token(T_ADD);
                        right = term();

                        /* the first term to be added is 'term_expr'.
                         * the second term is 'right'.
                         */

                        node = makeADDnode(term_expr, right);
                        term_expr = node;
                }

                else if (curr_token == T_SUB) {

                        match_token(T_SUB);
                        right = term();

                        /* we are subtracting the second term ('right')
                         * from the 'term_expr'.
                         */

                        node = makeSUBnode(term_expr, right);
                        term_expr = node;
                }
                else parse_error(); /* we should never get here. this is
                                     * overkill but lets make checking a habit.
                                     */

        }       /* while */

        if (DEBUG) printf(" -> end of expr()\n");
        return(term_expr);

}       /* expr() */



/* The addop() function performs the recursion for the BNF rule:
 *      addop -> ADD
 *               | SUB
 *
 * This ammounts to just checking for syntax errors and eating up a token.
 */
struct stree_node * addop() {

        if (DEBUG) printf("addop(): token = %s\n", yytext);

        if (curr_token == T_ADD) {
                match_token(T_ADD);

                if (DEBUG) printf(" -> end of addop()\n");
                return(0);
        }

        else if (curr_token == T_SUB) {
                match_token(T_SUB);

                if (DEBUG) printf(" -> end of addop()\n");
                return(0);
        }
```

```
        else
                parse_error();

        if (DEBUG) printf(" -> end of addop()\n");
        return(NULL);
}



/* The term() function performs the recursion for the BNF rule:
 *      term -> term mulop factor
 *              | factor
 *
 * This is done via an equivalent EBNF rule:
 *      term -> factor { mulop factor }
 */
struct stree_node * term() {

        /* pointers to sub trees. */
        struct stree_node *factor_expr;
        struct stree_node *node;
        struct stree_node *right;

        if (DEBUG) printf("term(): token = %s\n", yytext);

        factor_expr = factor();

        while (curr_token == T_MUL  ||  curr_token == T_DIV) {

                if (curr_token == T_MUL) {
                        if (DEBUG) printf(" -> MUL token\n");

                        match_token(T_MUL);
                        right = factor();

                        /* we are multipling 'factor_expr' and 'right'. */

                        node = makeMULnode(factor_expr, right);
                        factor_expr = node;
                }

                else if (curr_token == T_DIV) {
                        if (DEBUG) printf(" -> DIV token\n");

                        match_token(T_DIV);
                        right = factor();

                        /* we are dividing 'factor_expr' by 'right'. */

                        node = makeDIVnode(factor_expr, right);
                        factor_expr = node;
                }
                else parse_error(); /* we shouldn't be able to get here. */

        }       /* while */

        if (DEBUG) printf(" -> end of term()\n");

        return(factor_expr);

}       /* termP() */


/* The mulop() function performs the recursion for the BNF rule:
 *      mulop -> MUL
```

```
 *              | DIV
 *
 * This ammounts to just checking for syntax errors and eating up a token.
 */
struct stree_node * mulop() {

        if (DEBUG) printf("mulop(): token = %s\n", yytext);

        if (curr_token == T_MUL) {
                match_token(T_MUL);

                if (DEBUG) printf(" -> end of mulop()\n");
                return(0);
        }

        else if (curr_token == T_DIV) {
                match_token(T_DIV);

                if (DEBUG) printf(" -> end of mulop()\n");
                return(0);
        }

        else
                parse_error();

        if (DEBUG) printf(" -> end of mulop()\n");
        return(NULL);
}


/* The factor() function performs the recursion for the BNF rule:
 *      factor -> LPAR expr RPAR
 *              | ID
 *              | NUM
 *              | SUB NUM
 *
 * by examining the first (ie current) token to split up the rule into
 * four smaller rules.
 *
 */
struct stree_node * factor() {

        /* pointers to sub tree structures */
        struct stree_node *ret_tree;
        struct stree_node *an_expr;
        struct stree_node *ident_node;
        /* temporary holders for node infromation */
        char *ident;
        char *num;

        if (DEBUG) printf("factor(): token = %s\n", yytext);

        switch(curr_token) {

                case (T_LPAR):
                /* factor -> LPAR expr RPAR */

                        if (DEBUG) printf("LPAR type\n");
                        match_token(T_LPAR);
                        an_expr = expr();
                        match_token(T_RPAR);

                        if (DEBUG) printf(" -> end of factor()\n");
                        return(an_expr);
                        break;
```

```
                case (T_ID):
                /* factor -> ID */

                        if (DEBUG) printf("ID type\n");

                        ident_node = makeIDnode(yytext);
                        match_token(T_ID);

                        if (DEBUG) printf(" -> end of factor()\n");
                        return(ident_node);
                        break;

                case (T_NUM):
                /* factor -> NUM */

                        if (DEBUG) printf("NUM type\n");

                        ret_tree = makeNUMnode(yytext);
                        match_token(T_NUM);

                        if (DEBUG) printf(" -> end of factor()\n");
                        return(ret_tree);
                        break;

                case (T_SUB):
                /* factor -> SUB NUM */

                        if (DEBUG) printf("SUB type\n");
                        match_token(T_SUB);

                        /* put a negation sign on the string */
                        num = (char*) malloc(strlen(yytext)+2);
                        sprintf(num, "-%s", yytext);

                        match_token(T_NUM);

                        ret_tree = makeNUMnode(num);

                        free(num);
                        if (DEBUG) printf(" -> end of factor()\n");
                        return(ret_tree);
                        break;
                default:
                        parse_error();
                        break;



        }       /* switch */

}       /* factor() */
```