CPSC 411 ASSIGNMENT # 1
Winter 2001
Lec ??
Lab 02

Student: Chad C. D. Clark < clarkch @ cpsc . ucalgary . ca >
Date: Jan 28, 2002

Synopsis: A minusculus compiler written in pure C that generates stack
machine code as defined in file AS1.SPECS.

NOTE: I did use the _OLD_ grammar. Though the rules do get manipulated a
bit. Refer to source code comments for term, expr and, stmtlist.

Input: Reads minusculus source code from STDIN.
Output: Writes stack machine target code to STDOUT.
Errors (including warnings) are reported to STDERR.

Files:
README - contains info about this project and tarball.
AS1.SPECS - copy of the assignment in text format.
AS1.GRAMMAR - new grammar (non left-recursive).
as1lex.l - the lexer and main() routine.
as1parse.c - performs the parsing and syntax tree construction.
as1tree.c - function that deal with building the tree and traversing.
as1globals.h - info used by more than one file.
as1tokens.h - defines the token types used by the lexer and parser.
as1tree.h - defines node types and tree functions.
makefile - pretty standard makefile.
tests/ - directory with many test files.
tests/do-tests - script to run the tests found in tests directory.

Compiling:
Just use the command "make".
"make clean" deletes lex.yy.c, *.o and a.out
"make stats" counts source file lines.

Requires flex and gcc
- Developed with:
  - flex 2.5.4
  - egcs 2.91.66
  - gdb 4.18

Postmortum: (Mostly for my benifit but I wanted to keep this next to the code.)

Things that went well:
- Writing the tree pointer code. Only one null pointer error!
  It was due to not checking for NULL and not having an END_NODE.
- Writing the lexer. LEX rocks!

Things that went not so well:
- Getting the shape of the tree stucture down in my mind. I was
  starting to get two disjoint models (parse code and node building
  code) then I sat down, drew pictures and, traced out tree traversals
  untill I had a model. Then writing down a record of the model for
  reference.
- Realizing C unlike C++ requires ’struct’ before every declaration.

Things that were usefull:
- Drawing pictures (trees in particular).
- Writing down plans, decisions and models for reference in other
  parts of the program.
- Writing code on paper before typing it up.
- GDB and the GDB book by Stallman and Pesch.

- syntax highlighting & 60 row console mode

Things I would like to do differently on future projects:
- Sit down and plan out the structure of the program before writing code.
- Document my data structures before using them. It’s easier than searching with grep and opening up an editor.

Things I learned:
- A nice (ie non left recursive grammar) is very usefull.
- Follow sets can make parsing much easier.

Things I want to learn:
- How to let flex let me define main() outside of the .l file.

CPSC 411 Compiler construction I
Author: Robin Cockett
Date: 9 Jan. 2002

Assignment #1: Compiler for Minusculus (see link for grammar with left recursions removed)
Due date: 29th Jan 2002 (modified from 28 January 2002 to equalize labs.)

Implement a compiler to "stack machine code" (described below) for the language Minusculus whose syntax is defined by the grammar below.
In this assignment you may use LEX (in fact you MUST use LEX) but please write a recursive descent parser (i.e. you cannot use YACC yet!). It is important to attend the labs as they will be discussing this assignment and will also introduce LEX.

Minisculus grammar:

```
prog -> stmt.
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.
stmtlist -> stmtlist stmt SEMICOLON
            |.
expr -> expr addop term
            | term.
addop -> ADD
            | SUB.
term -> term mulop factor
            | factor.
mulop -> MUL
            | DIV.
factor -> LPAR expr RPAR
            | ID
            | NUM
            | SUB NUM.
```

This grammar has left recursive nonterminals. The grammar with the left recursions removed automatically is here . Notice this file calculates the first sets and follow sets of the above grammar and then transforms the grammar and introduces new nonterminals in this process.
Where the tokens above stand for:

```
"if" => IF
"then" => THEN
"while" => WHILE
"do" => DO
"until" => UNTIL
"read" => READ
"else" => ELSE
"begin" => BEGIN
"end" => END
"print" => PRINT
{alpha}[{digit}{alpha}]* => ID (identifier)
{digit}+ => NUM (positive integer)
"+" => ADD
"-" => SUB
"*" => MUL
"/" => DIV
```

```
  "(" => LPAR
  ")" => RPAR
  ";"=> SEMICOLON
```

```
Minisculus comments:
Minusculus has two types of comments:
  * multi-line comments:        /* comment */
  * and one line comments:      % comment
```

EXAMPLES:

Minsiculus program:
Here is an example program:

```
  /* This program calculates the factorial of the number input */
  begin % input a number ..

  read x;
   y:= 1;
   while x do
  begin

  y:= y * x;
  x:= x - 1;

  end;
  print y;

  end
```

Code generation:
Typical minusculus programs fragments are:

```
begin

  y:= 23;
  x: 13 + y;
  print x;

end

begin

  if y then x:= 10
         else x:= 1;
  z:= z * x

end
```

(where we assume here that the variables x, y, and z must have been initialized earlier: Note that for conditionals and while statements zero is false anything else is true). These fragments are translated into a stack machine code:

```
  cPUSH 23
  LOAD y
  cPUSH 13
  rPUSH y
  OP2 +
  LOAD x
  rPUSH x
```

```
  PRINT

  rPUSH y
  cJUMP   L1
  cPUSH 10
  LOAD x
  JUMP   L2

L1:

  cPUSH 1
  LOAD x

L2:

  rPUSH z
  rPUSH x
  OP2 *
  LOAD z
```

where

* cPUSH k --- push constant k onto stack
* rPUSH r --- push contents of register r onto stack
* sPUSH --- replaces the top element of the stack by the element it indexes in the stack
* LOAD r --- pop the top of the stack and put the value in register r
* OPn?? --- perform the operation on the top n values of the stack replacing them by the result
* cJUMP L --- conditional goto L (a label) pops top of stack and if it is zero (false) it jumps to label
* JUMP L --- unconditional jump to label
* PRINT --- pops and prints the top element of the stack
* READ r --- reads a value into register r (actually it reads a line and uses the first value on the line ...)

Here is an implementation of this stack machine in the shell: source this then source the file your Minusculus compiler produces!

```
  # ... aliases for Robin's stack machine.
  # This file should be "sourced" prior to executing
  # stack machine files.
  set stack = ""
  alias cPUSH            'set stack = (\!:1 $stack)'
  alias rPUSH            'set stack = ($\!:1 $stack)'
  alias sPUSH            '@ stack[1] = $stack[1] + 1 ; set stack[1] =
  $stack[$stack[1]]'
  alias LOAD            'eval "set \!:1 = \$stack[1] ; shift stack"'
  alias OP2               'eval "@ stack[2] = \$stack[2] \!:1
  \$stack[1]"; shift stack'
  alias cJUMP           'set tos = $stack[1]; shift stack; if ($tos ==
  0) goto \!:1'
  alias JUMP             goto
  alias PRINT            'echo $stack[1]; shift stack'
  alias READ            'eval "set \!:1 = $< " '
```

```
Context Free Grammar Vital Statistics Checker (version 0.00)
---- Please report bugs and problems to robin@cpsc
============================================================

Your grammar is:


prog ->  stmt.

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.

stmtlist ->  stmtlist stmt SEMICOLON
           |.

expr ->  expr addop term
       | term.

addop ->  ADD
        | SUB.

term ->  term mulop factor
       | factor.

mulop ->  MUL
        | DIV.

factor ->  LPAR expr RPAR
         | ID
         | NUM
         | SUB NUM.


All nonterminals are reachable and realizable.


The nullable nonterminals are:

 stmtlist.

The endable nonterminals are:

 factor term expr prog stmt.

The following nonterminals are left recursive:
 stmtlist expr term

The grammar is not LL(1).



The first sets are:

stmt ====> { IF WHILE DO READ ID PRINT BEGIN }
addop ====> { ADD SUB }
mulop ====> { MUL DIV }
factor ====> { LPAR ID NUM SUB }
prog ====> { IF WHILE DO READ ID PRINT BEGIN }
stmtlist ====> { IF WHILE DO READ ID PRINT BEGIN }
term ====> { LPAR ID NUM SUB }
```

```
expr ====> { LPAR ID NUM SUB }


The follow sets are:

mulop ====> { LPAR ID NUM SUB }
factor ====> { MUL DIV ELSE UNTIL SEMICOLON THEN DO ADD SUB RPAR }
addop ====> { LPAR ID NUM SUB }
term ====> { MUL DIV ELSE UNTIL SEMICOLON THEN DO ADD SUB RPAR }
stmtlist ====> { END IF WHILE DO READ ID PRINT BEGIN }
expr ====> { ELSE UNTIL SEMICOLON THEN DO ADD SUB RPAR }
stmt ====> { ELSE UNTIL SEMICOLON }




=====================================================================

                TRANSFORMING THE GRAMMAR
                to remove left recursion.

=====================================================================

Your grammar is:


prog ->  stmt.

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.

stmtlist ->  stmtlist+.

stmtlist+ ->  stmt SEMICOLON stmtlist+
            |.

expr ->  term expr+.

expr+ ->  addop term expr+
        |.

addop ->  ADD
        | SUB.

term ->  factor term+.

term+ ->  mulop factor term+
        |.

mulop ->  MUL
        | DIV.

factor ->  LPAR expr RPAR
         | ID
         | NUM
         | SUB NUM.


All nonterminals are reachable and realizable.
```

The nullable nonterminals are:

stmtlist+ expr+ term+ stmtlist.

The endable nonterminals are:

term+ factor expr+ term expr prog stmt.

The grammar has no left recursive nonterminals.

The grammar is LL(1).

The first sets are:

```
stmt ====> { IF WHILE DO READ ID PRINT BEGIN }
addop ====> { ADD SUB }
mulop ====> { MUL DIV }
factor ====> { LPAR ID NUM SUB }
prog ====> { IF WHILE DO READ ID PRINT BEGIN }
stmtlist+ ====> { IF WHILE DO READ ID PRINT BEGIN }
expr+ ====> { ADD SUB }
term ====> { LPAR ID NUM SUB }
term+ ====> { MUL DIV }
stmtlist ====> { IF WHILE DO READ ID PRINT BEGIN }
expr ====> { LPAR ID NUM SUB }
```

The follow sets are:

```
mulop ====> { LPAR ID NUM SUB }
term+ ====> { ADD SUB ELSE UNTIL SEMICOLON THEN DO RPAR }
factor ====> { MUL DIV ADD SUB ELSE UNTIL SEMICOLON THEN DO RPAR }
addop ====> { LPAR ID NUM SUB }
expr+ ====> { ELSE UNTIL SEMICOLON THEN DO RPAR }
term ====> { ADD SUB ELSE UNTIL SEMICOLON THEN DO RPAR }
stmtlist ====> { END }
stmtlist+ ====> { END }
expr ====> { ELSE UNTIL SEMICOLON THEN DO RPAR }
stmt ====> { ELSE UNTIL SEMICOLON }
```

```
a.out: lex.yy.c as1pars.o as1tree.o
	gcc lex.yy.c -ll as1pars.o as1tree.o -g

lex.yy.c: as1lex.l
	flex as1lex.l

as1pars.o: as1pars.c as1globals.h as1tokens.h
	gcc -g -c as1pars.c -o as1pars.o

as1tree.o: as1tree.h as1tree.c
	gcc -g -c as1tree.c -o as1tree.o

clean:
	rm -rf lex.yy.c *.o a.out

stats:
	make clean;
	cat *.[lhc] | wc
```

```
%{

/* chad c d clark   < clarkch @ cpsc . ucalgary . ca >
 *
 * cpsc 411     lec ??
 * winter 2002  lab 02
 *
 * assignment #1 - a first stab.
 *
 * file: as1lex.l
 * purpose: a basic lexer
 *
 * assumtions:
 *    - all keywords are lowercase.
 *    - identifiers are case sensitive, start with an alphabetic character,
 *      and consist of alphabetic and numeric characters.
 *    - whitespace is to be ignored. (syntax oriented, not line oriented.)
 *
 */

#include "as1tokens.h"
#include "as1globals.h"
#include "as1tree.h"


%}


IF      "if"
THEN    "then"
WHILE   "while"
DO      "do"
UNTIL   "until"
READ    "read"
ELSE    "else"
BEGIN   "begin"
END     "end"
PRINT   "print"
ID      [a-zA-Z][0-9a-zA-Z]*
NUM     [0-9]+
ADD     "+"
SUB     "-"
MUL     "*"
DIV     "/"
LPAR    "("
RPAR    ")"
SEMICOLON       ";"
ASSIGN  ":="
MULTISTART      "/*"
MULTIEND        "*/"


%x COMMENT

%%

{MULTISTART}            {BEGIN COMMENT;}
<COMMENT>{MULTIEND}     {BEGIN 0;}
<COMMENT>\n             { /* multiline comment */ }
<COMMENT>.              { /* multiline comment */ }

"%".*\n         { /* single line comment */ }

[ ]     { /* whitespace */ }
\t      { /* whitespace */ }
```

```
\n      { /* whitespace */ }

{IF}    { return (T_IF); }
{THEN}  { return (T_THEN); }
{WHILE} { return (T_WHILE); }
{DO}    { return (T_DO); }
{UNTIL} { return (T_UNTIL); }
{READ}  { return (T_READ); }
{ELSE}  { return (T_ELSE); }
{BEGIN} { return (T_BEGIN); }
{END}   { return (T_END); }
{PRINT} { return (T_PRINT); }
{ID}    { return (T_ID); }
{NUM}   { return (T_NUM); }
{ADD}   { return (T_ADD); }
{SUB}   { return (T_SUB); }
{MUL}   { return (T_MUL); }
{DIV}   { return (T_DIV); }
{LPAR}  { return (T_LPAR); }
{RPAR}  { return (T_RPAR); }
{SEMICOLON}     { return (T_SEMICOLON); }
{ASSIGN}        { return (T_ASSIGN); }
.       { return (T_ERROR); }

%%


/* chad c d clark   < clarkch @ cpsc . ucalgary . ca >
 *
 * cpsc 411     lec ??
 * winter 2002  lab 02
 *
 * assignment #1 - a first stab.
 *
 * function: as1main.c
 * purpose: tests the lexer's return values.
 *
 *
 */

struct stree_node * prog(void);

int main(int argc, char **argv) {

        struct stree_node *STree = NULL;

        if(!FINAL) printf("\n");

        curr_token = yylex();

        STree = prog();

        if(PRINT_STREE) {       /* see as1globals.h */
                printf("\nSyntax Tree:\n");
                print_stree(STree, 0);
        }

        /*
        * printf("\n\nSeaching for PRINT_NODE's ... ");
        * if (find_node(PRINT_NODE, STree))
        *       printf("success!\n");
        * else
        *       printf("failed!\n");
        */
```

```
        if(!FINAL) printf("\nGenerated Code:\n");
        gen_code(STree);

        delete_stree(STree);

        if(!FINAL) printf("\n");
        return(0);
}
```

```
/* chad c d clark   < clarkch @ cpsc . ucalgary . ca >
 *
 * cpsc 411     lec ??
 * winter 2002  lab 02
 *
 * assignment #1 - a first stab.
 *
 * file: as1tokens.h
 * purpose: token defines for lexing and parsing.
 *
 *
 */


#define T_IF    240
#define T_THEN  241
#define T_ELSE  242
#define T_WHILE 230
#define T_DO    231
#define T_UNTIL 232
#define T_READ  220
#define T_PRINT 221
#define T_BEGIN 210
#define T_END   211
#define T_ID    200
#define T_NUM   201
#define T_ADD   190
#define T_SUB   191
#define T_MUL   192
#define T_DIV   193
#define T_LPAR  180
#define T_RPAR  181
#define T_SEMICOLON     182
#define T_ASSIGN        183
#define T_ERROR 255
```

```
/* chad c d clark   < clarkch @ cpsc . ucalgary . ca >
 *
 * cpsc 411     lec ??
 * winter 2002  lab 02
 *
 * assignment #1 - a first stab.
 *
 * file: as1globals.h
 * purpose: defines the global variables to be used.
 *
 *
 */



/* this holds the type of token we are currently looking at
 * (ie dealing with).
 *
 */
int curr_token = -1;

/* this is used for the labels in the code generation stage. */
int label_counter = 1;

/* set PRINT_STREE to 1 to have main print out the tree
 * set PRINT_STREE to 0 to have main not print out the tree
 *
 */
#define PRINT_STREE 0

/* set FINAL to 1 to turn off some usefull printf()'s that we don't
 * want to see in the final output of this project.
 * set it to zero otherwize (when developing).
 *
 */
#define FINAL 1
```

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

#define IF_NODE         501
#define WHILE_NODE      502
#define DO_NODE         503
#define ASSIGN_NODE     504
#define READ_NODE       505
#define PRINT_NODE      506
#define BEGIN_NODE      507
#define STMTLIST_NODE   508
#define ADD_NODE        509
#define SUB_NODE        510
#define MUL_NODE        511
#define DIV_NODE        512
#define ID_NODE         513
#define NUM_NODE        514


struct stree_node {

        /* type of node this is */
        int type;

        /* the next argument for a parent node */
        struct stree_node * sibling;
        /* the sub-tree link */
        struct stree_node * child;

        /* numeric value of this node */
        int numval;
        /* string value of this node (used for identifier names) */
        char * idval;
};




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

struct stree_node * makeIFnode(struct stree_node *if_expr,
                struct stree_node *true_stmt,
                struct stree_node *false_stmt);


struct stree_node * makeWHILEnode(struct stree_node *while_expr,
                struct stree_node *do_stmt);


struct stree_node * makeDOnode(struct stree_node *do_stmt,
                struct stree_node *while_expr);


struct stree_node * makeASSIGNnode(struct stree_node *id_node,
```

```
                struct stree_node *an_expr);


struct stree_node *makeREADnode(struct stree_node *id_node);


struct stree_node *makePRINTnode(struct stree_node *an_expr);


struct stree_node *makeBEGINnode(struct stree_node *slist);


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

struct stree_node *makeSTMTLISTnode(struct stree_node *a_stmt,
                struct stree_node *a_stmtlist);

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

struct stree_node *makeADDnode(struct stree_node *a_term,
                struct stree_node * right_term);

struct stree_node *makeSUBnode(struct stree_node *a_term,
                struct stree_node * right_term);


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

struct stree_node *makeMULnode(struct stree_node *a_factor,
                struct stree_node * right_factor);


struct stree_node *makeDIVnode(struct stree_node *a_factor,
                struct stree_node * right_factor);




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

struct stree_node *makeIDnode(char *ident);


struct stree_node *makeNUMnode(char *num);


/* function to print out the syntax tree *******************/
void print_stree(struct stree_node *node, int spaces);

/* function to find a type of node in the tree *************/
int find_node(int type, struct stree_node *node);

/* function to recursivly delete a tree ********************/
void delete_stree(struct stree_node *node);
```

```
/* 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() */
```

```
/* 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() */
```