The Advanced Calculator Lexer

The six comparison operators all return a CMP token with a lexical value to distinguish them.
The six keywords and four built-in functions are recognized by literal patterns.

Note that they have to precede the general pattern to match a name so that they’re matched in preference to the general pattern.

The name pattern looks up the name in the Symbol table and returns a pointer to the symbol.
/* debug hack */
"debug"[0-9]+ { debug = atoi(&yytext[5]); printf("debug set to %d\n", debug); }

/* names */
[a-zA-Z][a-zA-Z0-9]* { yylval.s = lookup(yytext); return NAME; }
[0-9]+\."[0-9]*\{EXP}\? |
\."?[0-9]+\{EXP}\? { yylval.d = atof(yytext); return NUMBER; }

"//".*
[ \t] /* ignore white space */
\\n printf("c> "); /* ignore line continuation */
\\n { return EOL; }
. { yyerror("Mystery character %c\n", *yytext); }
%`
The heart of the calculator is eval, which evaluates an AST built up in the parser. Following the practice in C, comparisons return 1 or 0 depending on whether the comparison succeeds, and tests in if/then/else and while/do treat any nonzero as true.
/* control flow */
/* null if/else/do expressions allowed in the grammar, so check for them */
case 'I':
  if (eval(((struct flow *)a)->cond) != 0) {
    if ( ((struct flow *)a)->tl) {
      v = eval(((struct flow *)a)->tl);
    } else
      v = 0.0;            /* a default value */
  } else {
    if ( ((struct flow *)a)->el) {
      v = eval(((struct flow *)a)->el);
    } else
      v = 0.0;            /* a default value */
  }
  break;

case 'W':
  v = 0.0;            /* a default value */

  if ( ((struct flow *)a)->tl) {
    while( eval(((struct flow *)a)->cond) != 0)
      v = eval(((struct flow *)a)->tl);
  } break;        /* last value is value */

case 'L': eval(a->l); v = eval(a->r); break;

case 'F': v = callbuiltin((struct fncall *)a); break;

case 'C': v = calluser((struct ufnccall *)a); break;

default: printf("internal error: bad node %c\n", a->nodetype);
} return v;
For expressions, we do the familiar depth-first tree walk to compute the value.

An AST makes it straightforward to implement if/then/else: Evaluate the condition AST to decide which branch to take, and then evaluate the AST for the path to be taken.

To evaluate while/do loops, a loop in eval evaluates the condition AST, then the body AST, repeating as long as the condition AST remains true.

Any AST that references variables that are changed by an assignment will have a new value each time it’s evaluated.
Built-in functions are relatively straightforward: Determine which function it is and call specific code to do the function.

```c
static double
callbuiltin(struct fn* f)
{
    enum bifs functype = f->functype;
    double v = eval(f->l);

    switch(functype) {
    case B_sqrt:     
        return sqrt(v);
    case B_exp:     
        return exp(v);
    case B_log:     
        return log(v);
    case B_print:  
        printf(“= %4.4g\n”, v);
        return v;
    default:  
        yyerror("Unknown built-in function \%d", functype);
        return 0.0;
    }
}
A function definition consists of the name of the function, a list of dummy Arguments, and an AST that represents the body of the function.

Defining the function simply saves the argument list and AST in the function’s symbol table entry, replacing any previous version.

```c
/* define a function */
void
dodef(struct symbol *name, struct symlist *syms, struct ast *func)
{
    if(name->syms) symlistfree(name->syms);
    if(name->func) treefree(name->func);
    name->syms = syms;
    name->func = func;
}
```
Say you define a function to calculate the maximum of its two arguments:

```plaintext
> let max(x,y) = if x >= y then x; else y;;
> max(4+5,6+7)
```

The function has two dummy arguments, x and y. When the function is called, the evaluator does this:

1. Evaluate the actual arguments, 4+5 and 6+7 in this case.
2. Save the current values of the dummy arguments and assign the values of the actual arguments to them.
3. Evaluate the body of the function, which will now use the actual argument values when it refers to the dummy arguments.
4. Put back the old values of the dummies.
5. Return the value of the body expression.
static double
calluser(struct ufncall *f)
{
    struct symbol *fn = f->s;   /* function name */
    struct symlist *sl;         /* dummy arguments */
    struct ast *args = f->l;    /* actual arguments */
    double *oldval, *newval;   /* saved arg values */
    double v;
    int nargs;
    int i;

    if(!fn->func) {
        yyerror("call to undefined function", fn->name);
        return 0;
    }

    /* count the arguments */
    sl = fn->syms;
    for(nargs = 0; sl; sl = sl->next)
        nargs++;

    /* prepare to save them */
    oldval = (double *)malloc(nargs * sizeof(double));
    newval = (double *)malloc(nargs * sizeof(double));
    if(!oldval || !newval) {
        yyerror("Out of space in %s", fn->name); return 0.0;
    }

    /* evaluate the arguments */
    for(i = 0; i < nargs; i++) {
        if(!args) {
            yyerror("too few args in call to %s", fn->name);
            free(oldval); free(newval);
            return 0;
        }

        if(args->nodetype == 'L') { /* if this is a list node */
            newval[i] = eval(args->l);
            args = args->r;
        } else { /* if it's the end of the list */
            newval[i] = eval(args);
            args = NULL;
        }
    }
}
/* save old values of dummies, assign new ones */
sl = fn->syms;
    for(i = 0; i < nargs; i++) {
        struct symbol *s = sl->sym;

        oldval[i] = s->value;
        s->value = newval[i];
        sl = sl->next;
    }
free(newval);

/* evaluate the function */
v = eval(fn->func);

/* put the dummies back */
sl = fn->syms;
    for(i = 0; i < nargs; i++) {
        struct symbol *s = sl->sym;

        s->value = oldval[i];
        sl = sl->next;
    }
free(oldval);
 return v;