An Advanced Calculator

The final example extends the calculator to make it a small but somewhat realistic “compiler”.

We’ll add
named variables and assignments;
comparison expressions (greater, less, equal, etc.);
flow control with if/then/else and while/do;
built-in and user-defined functions;
and a little error recovery.

The previous version of the calculator didn’t take much advantage of the AST representation of expressions, but in this one, the AST is the key to the implementation of flow control and user functions.
Here's an example of defining a user function, and then calling it, using a built-in function as one of the arguments:

> let avg(a,b) = (a+b)/2;
Defined avg
> avg(3, sqrt(25))
= 4
In mathematics, the Euclidean algorithm, or Euclid's algorithm, is an efficient method for computing the greatest common divisor (GCD) of two numbers, the largest number that divides both of them without leaving a remainder.

It is named after the ancient Greek mathematician Euclid, who first described it in his Elements (c. 300 BC).

It is an example of an algorithm, a step-by-step procedure for performing a calculation according to well-defined rules, and is one of the oldest algorithms in common use.
let euclid(x,y) = \\
    if (x == y) then x; \\
    else if (x > y) then euclid(x-y, y); \\
    else euclid (x, y-x);\ 
;;

Defined euclid

euclid(9,12)
euclid(12,20)
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.
Advanced Calculator - HASHING

Assume that you have an object and you want to assign a key to it to make searching easy.

To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values.

However, in cases where the keys are large and cannot be used directly as an index, you should use hashing.

In hashing, large keys are converted into small keys by using hash functions.

The values are then stored in a data structure called hash table.

The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key).

By using that key you can access the element in $O(1)$ time.

Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.
An Advanced Calculator - Symbol Table

The hash function is also quite simple: For each character, multiply the previous hash by 9 and then xor the character.

```c
/* hash a symbol */
static unsigned
symhash(char *sym)
{
    unsigned int hash = 0;
    unsigned c;

    while(c = *sym++) hash = hash*9 ^ c;

    return hash;
}
```

```c
/* symbol table */
struct symbol {
    /* a variable name */
    char *name;
    char *name;
    double value;
    struct ast *func; /* stmt for the function */
    struct symlist *syms; /* list of dummy args */
};
```
The lookup routine computes the symbol table entry index as the hash value modulo the size of the symbol table, which was chosen as a number with no even factors, again to mix the hash bits up.

lookup takes a string and returns the address of the table entry for that name, creating a new entry if there isn't one already. The lookup technique is known as hashing with linear probing. It uses a hash function to turn the string into an entry number in the table, then checks the entry, and, if it's already taken by a different symbol, scans linearly until it finds a free entry.
There are two ways to specify precedence and associativity in a grammar, implicitly and explicitly.

So far, we’ve specified them implicitly, by using separate non-terminal symbols for each precedence level.

This is a perfectly reasonable way to write a grammar, and if bison didn’t have explicit precedence rules, it would be the only way.
The `%union` here defines many kinds of symbol values, which is typical in realistic bison parsers.

As well as a pointer to an AST and a numeric value, a value can be a pointer to the symbol table for a user symbol, a list of symbols, or a subtype of a comparison or function token.

(We use the word symbol somewhat confusingly here, both for names used in the bison grammar and for names that the user types into the compiled program.

We'll say user symbol for the latter when the context isn't otherwise clear.)
Each of these declarations defines a level of precedence, with the order of the %left, %right, and %nonassoc declarations defining the order of precedence from lowest to highest. The definition of non-associative operators is that it is illegal to combine two or more of these without explicit parentheses.
Error parsing grammar go
To EOL

Built in
The rule for negation includes %prec UMINUS.

The only operator in this rule is -, which has low precedence, but we want unary minus to have higher precedence than multiplication rather than lower.

The %prec tells bison to use the precedence of UMINUS for this rule.
In the calculator, each symbol can potentially be both a variable and a user-defined function.

The value field holds the symbol’s value as a variable, the func field points to the AST for the user code for the function, and syms points to a linked list of the dummy (formal) arguments, which are themselves symbols. (In the previous example, avg is the function, and a and b are the dummy arguments.)

The C functions newsymlist and symlistfree create and free them.
Our grammar distinguishes between statements (stmt) and expressions (exp).

A statement is either a flow of control (if/then/else or while/do) or an expression.

The if and while statements take lists of statements, with each statement in the list being followed by a semicolon.

Each rule that matches a statement calls a routine to build an appropriate AST node.

```c
struct flow {
    int nodetype;     /* type I or W */
    struct ast *cond; /* condition */
    struct ast *tl;   /* then or do list */
    struct ast *el;   /* optional else list */
};

struct ast *
newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *el)
{
    struct flow *a = malloc(sizeof(struct flow));

    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = nodetype;
    a->cond = cond;
    a->tl = tl;
    a->el = el;
    return (struct ast *)a;
}
```
struct symref {
    int nodetype;    /* type N */
    struct symbol *s;
};

struct symasgn {
    int nodetype;    /* type = */
    struct symbol *s;
    struct ast *v;    /* value */
};
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.
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.
/* 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 fn呼叫 *)a); break;

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

default: printf("internal error: bad node %s\n", a->nodetype); 
} return v;
Built-in functions are relatively straightforward: Determine which function it is and call specific code to do the function.

```c
static double
callbuiltin(struct fncall *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;
}
```
```c
struct ast *
newfunc(int functype, struct ast *l)
{
    struct fncall *a = malloc(sizeof(struct fncall));

    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = 'F';
    a->l = l;
    a->functype = functype;
    return (struct ast *)a;
}
```
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 ufuncall *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->symns;
    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->sym;
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->sym;
for(i = 0; i < nargs; i++) {
    struct symbol *s = sl->sym;
    s->value = oldval[i];
    sl = sl->next;
}
free(oldval);
return v;