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:

```plaintext
> let avg(a,b) = (a+b)/2;
Defined avg
> avg(3, sqrt(25))
= 4
```
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.
The hash function is also quite simple: For each character, multiply the previous hash by 9 and then xor the character.
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.
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.

The C functions newsymlist and symlistfree create and free them.

```c
/* list of symbols, for an argument list */
struct symlist {
  struct symbol *sym;
  struct symlist *next;
};

struct symlist *newsymlist(struct symbol *sym, struct symlist *next);
void symlistfree(struct symlist *sl);```
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 nonterminal 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.
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.
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.)
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) {
        yynerror("out of space");
        exit(0);
    }
    a->nodetype = nodetype;
    a->cond = cond;
    a->tl = tl;
    a->el = el;
    return (struct ast *)a;
}
```
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.