```
exp       ::= value
            | variable-name
            | (set variable-name exp)
            | (if exp exp exp)
            | (while exp exp)
            | (begin {exp})
            | (function {exp})

formals   ::= {variable-name}

value     ::= integer

function  ::= function-name
            | primitive

primitive ::= + | - | * | / | = | < | > | print

integer   ::= sequence of digits, possibly prefixed with a plus or minus sign

*-name    ::= sequence of characters not an integer and not containing (, ), ;,
              or whitespace
```

It is not useful to define a *function-name* that is one of the "keywords" `define`, `if`, `while`, `begin`, or `set`. It is not wise to define a *function-name* that is one of the *primitive*s. Aside from these restrictions, names can use any characters except parentheses, semicolon, whitespace, and nulls. A comment is introduced by a semicolon, and it continues to the end of the line; therefore a semicolon cannot occur within a name.

The definition form `(val x` *e*`)` defines a new global variable `x` and initializes it to the value of the expression *e*. Global variables must be defined before they are used or assigned to. The `define` form defines a new function. The form `(use` *file-name*`)` is not a true definition, but evaluating it causes the interpreter to read the definitions in the named file as if they had been typed directly to the interpreter. Finally, the `check-expect` and `check-error` forms are even less like the other definitions: they "define" tests that are run once the interpreter has finished with the file in which they appear.

Expressions are fully parenthesized. We use *prefix* syntax, in which each operator precedes its arguments. Thus, translating the C assignment `i = 2*j + i - k/3` produces the Impcore expression `(set i (- (+ (* 2 j) i) (/ k 3)))`. The Impcore syntax is trivial to parse; C syntax is anything but. Readers accustomed to *infix* syntax, in which binary operators appear between their arguments, may find prefix syntax unattractive, especially in complex expressions. Luckily, expressions even this complex occur rarely.

2.1.2 Meaning

I present the meanings of Impcore expressions using informal English, which is intuitive but not precise. Section 2.4 presents a precise, formal semantics.

In Impcore, all values are integers; as in C, `if` and `while` use their conditions by interpreting zero as false and nonzero as true.

`(if` e_1 e_2 e_3`)` — Evaluate e_1; if it is nonzero, evaluate e_2 and return the result, otherwise evaluate e_3 and return the result.

`(while` e_1 e_2`)` — Evaluate e_1; if it is zero, return zero; otherwise, evaluate e_2 and then re-evaluate e_1; continue until e_1 evaluates to zero. (A `while` expression always returns zero, but that doesn't matter since it is typically evaluated for its side effects.)