`input_object` deals with whole symbols: `next()` and `back()`

`parsing_object` deals with whole components of a program, all the way from simple expressions to statements to function definitions. Contains its own input object. Parsing functions return pointers to **`nodes`**: `read_simple_expr()`, `read_adding_expr()`, `read_statement()`, etc.

Syntax so far:

Simple Expression	SE	::=	identifier | number
Adding Expression	AE	::=	SE ((+ | -) SE)*
Statement	STMT	::=	`print` AE ;
Program	PROG	::=	STMT end-of-file

```
read_simple_expr():
        use next() to get one symbol
        if it is a number or identifier, create appropriate node and return pointer
        otherwise error message and return NULL.

read_adding_expr():
        use read_simple_expr() to get first component, save as L
        enter loop:
            use next() to get next symbol, save it as OP
            if it is not plus or minus:
                use back() so it can be seen again
                break out of loop
            use read_simple_expr() to get next component, save as R
            combine L, OP, R into single expression node, use as new L
        after loop:
            return L

read_statement():
        Use next() to get first symbol
        if it is "print":
            use read_adding_expr() to get expression, save as E
            use next() to check for semi-colon
            if semi-colon not found:
                error message, back(), return NULL;
            make a print-statement node containing E, return as result
        otherwise
            error message
            return NULL;

read_program():
        Use read_statement() to get result
        use next() to see final symbol
        if it is not end-of-file
            error message.
```

memory_object has the simple task of remembering the values of variables while a program is running. It needs a `get(string)` method to retrieve the value of a variable and a `set(string, int)` method to record a new value. For testing purposes, the implementations could be as simple as

```
int get(string varname)
{ if (varname==”x”)
    return 123;
  else if (varname==”y”)
    return 456;
  else
    return 789; }

void set(string varname, int value)
{ cout << “pretending to remember “ << varname << “ = “ << value << “\n”; }
```

A working memory_object could perhaps have a vector of strings and a vector of ints so that `set` can really do its job.

interpreting_object is responsible for executing the program once the parsing object has done its job and provided a pointer to the tree for the whole program. In interpreting object has its own memory_object as a member. There are two main methods:

`int interpreting_object::value_of(node * t)` given a pointer to a tree that represents some kind of expression, does whatever is required to find the value of that expression, which is returned as its result.

```
if t is NULL:
    error, give up
else if t->kind is “number”
    return the value of that number
else if t->kind is “identifier”
    use mem.get to find the identifier’s value, and return that.
else if t->kind is “expression”
    A = value_of(t->ptr1);
    B = value_of(t->ptr2);
    if detail is “+”
        return A+B
    else if detail is “-“
        return A-B
    else
        error
    A = value_of(t->ptr1);
else
    error
```

`void interpreting_object::execute(node * t)` given a pointer to a tree that represents some kind of statement, does whatever is required to produce the proper results from executing that statement.

```
if t is NULL:
    error, give up
else if t->kind is “print”
    A = value_of(t->ptr1);
    cout A
else
    error
```

As features are added to the language to make it less trivial, this basic framework is gradually expanded, but always keeps the same essential form.

The plan of a program that uses all of this to make the programming language usable is something like this:

```
initialise parsing_object po
initialise interpreting_object ex
node * prog = NULL
repeat
    ask user’s wishes
    if user selects “enter a new program”:
        prog = po.read_program();
    else if user selects “show the tree”:
        prog->print();
    else if user selects “run program”
        ex.execute(prog);
```