Adding an assignment statement to the syntax

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

Nothing changes except that an extra case is added to `read_statement` and `execute`.

```
read_statement():
        Use next() to get first symbol
        if it is "print":
                SAME AS BEFORE
        if it is an identifier:
                create a new node to represent this identifier, save it as ID
                    ( a good way to do this is to call back(), then let read_simple_expr() do
                      the real work for you )
                use next() to check for "=" symbol
                if there is no "=": error and return NULL
                use read_adding_expression() to read the AE, save it as VAL
                use next() to check for semi-colon
                if semi-colon not found:
                        error message, return NULL;
                make a assignment-statement node containing ID and VAL,
                return that node (pointer) as result
        otherwise
                SAME AS BEFORE

execute(node * t)
        if t is NULL:
                SAME AS BEFORE
        else if t->kind is "print"
                SAME AS BEFORE
        else if t->kind is "assignment"
                follow the pointers to get the variable name:
                        string varname = t->ptr1->detail
                use value_of to evaluate the expression
                        int val = value_of(t->ptr2)
                mem.set(varname, val)
        else
                SAME AS BEFORE
```

Allowing parentheses in expressions

Just realise that "(" followed by any expression, followed by ")" behaves like a very simple basic expression, so add one clause to SE:

Simple Expression	SE	::=	identifier
			| number
			| (number)

This is implemented by adding one new case to read_simple_expr(). After checking for an identifier or a number, check for an opening parenthesis:

```
use next() to get one symbol
if it is a number or identifier
            create appropriate node and return pointer
otherwise if it is "("
            call read_adding_expression to do its job, save result as E
            call next() to check for ")"
            if ")" not present, error message and return NULL
            otherwise return E
otherwise error message and return NULL.
```

No other additions are needed. Parentheses in expressions just change the way the parser builds the tree.

Defining a **block**, or sequence of statements, which now becomes the main thing in a program:

Block	BLOCK	::=	{ STMT * }
Program	PROG	::=	BLOCK end-of-file

This requires a new parsing method, perhaps called `read_block()`:

```
use next() to check for "{"
if "{" not present, error message, return NULL
L = NULL
enter loop:
    use next() to check for "}"
    if "}" is seen:
        break from loop.
    use read_statement() to read just one statement, save result as S
    if L is still NULL
        set L = S
    otherwise
        create new node labelled "sequence" with pointers L and S
        set L = that new node
after end of loop:
    if L is still NULL
        replace L with new node labelled "empty statement", no content
    return L as result.
```

Also add a case to execute() to handle these two new kinds of node:

```
if t->kind is "empty statement":
    don't do anything, the program was just "{ }".
if t->kind is "sequence":
    do the first step - execute(t->ptr1)
    do the second step - execute(t->ptr2)
    that's it.
```

To allow a block to appear as a kind of statement:

Statement	STMT	::=	print AE ;
		|	identifier = AE ;
		|	BLOCK

Fortunately, a block always begins with "{", which is distinct from the existing cases, so just add a new case to read_statement:

Use next() to get first symbol
if it is "print":
    SAME AS BEFORE
if it is an identifier:
    SAME AS BEFORE
if it is "{":
    use back(). The "{" is block's responsibility.
    call read_block(), return whatever it gives you.
otherwise
    SAME AS BEFORE

You may like to exercise your minds by thinking about how new operators may be added, such as *, /, <, >, etc., and then about how an if statement could be invented.