SeqExp

Syntax

Lexical

```
SEMI ';'
```

BNF

```
<exp>:SeqExp   ::= LBRACE <exp> <seqExps> RBRACE
<seqExps>      **= SEMI <exp>
```

Examples

```
% Must contain at least one expression.
{ 3 }

% Additional expressions are separated by ";".
{  3
;  +(2, 3)
}

% Each expression can be any expression (incomplete example).
{   x     % VarExp
;   3     % LitExp
;   +(4, x)   % PrimappExp
;   let ... in ...    % LetExp
;   proc ...          % ProcExp
;   . f (...) ...     % AppExp
;   { ... }           % SeqExp
}
```

Semantics

SeqExp

```
%%%
    // <exp>:SeqExp   ::= LBRACE <exp> <seqExps> RBRACE
    public Val eval(Env env) {
        Val v = exp.eval(env);          // (1)
        for (Exp e : seqExps.expList)
            v = e.eval(env);
        return v;                       // (2)
    }
%%%
```

1. Evaluate each expression in order in the containing environment.
2. The result of the SeqExp is the result of the last expression in the sequence.

Examples

```
{1 ; 3 ; 5}
% is 5

{42}
% is 42
```

Use

Until we have side-effects, the SeqExp is not very useful. Notice that the results of each expression except the last are simply ignored. Also notice that without side-effects, the expressions in the sequence cannot interfere with each other. So all but the last are expression are useless.

However, we can now use the SeqExp to add curly braces around any expression without changing the meaning of that expression, possibly improving the readability of expressions.

```
% Here we wrap the ProcExp in {}.
. {proc(t,u) +(t,u)} (3,4)


% We cannot put {} around the LetDecls because they are not an Exp.
% But we can around the RHS or parts of the RHS.
let

    double = proc(x) {
        *(2, x)
    }

    x = 8

in {
    let
        y = 4
    in {
        +(.double(x), y)
    }
}
% is 16
```