Languages-beta: SL-2-Expressions *

The PLanCompS Project

`SL-2-Expressions.cbs` | PLAIN | PRETTY

Language "SL"

2 Expressions

Syntax *Expr* : expr ::= int
| string
| 'true'
| 'false'
| expr '+' expr
| expr '/' expr
| expr '*' expr
| expr '-' expr
| expr '==' expr
| expr '<=' expr
| expr '<' expr
| expr '!=' expr
| expr '>=' expr
| expr '>' expr
| expr '&&' expr
| expr '||' expr
| '!' expr
| id '(' expr-list? ')'
| id
| id '=' expr
| expr '.' id
| expr '.' id '=' expr
| expr '.' id '(' expr-list? ')'
| '(' expr ')'

Rule ⟦ '(' *Expr* ')' ⟧ : expr = ⟦ *Expr* ⟧

Type sl-values ⇝ booleans | integers | strings | objects | null-type

*Suggestions for improvement: `plancomps@gmail.com`.
Reports of issues: `https://github.com/plancomps/CBS-beta/issues`.

```
Semantics  eval⟦ Expr : expr ⟧ : ⇒ sl-values
Rule       eval⟦ Int ⟧ = int-val⟦ Int ⟧
Rule       eval⟦ String ⟧ = string-val⟦ String ⟧
Rule       eval⟦ 'true' ⟧ = true
Rule       eval⟦ 'false' ⟧ = false
Rule       eval⟦ Expr1 '+' Expr2 ⟧ =
             integer-add-else-string-append(eval⟦ Expr1 ⟧, eval⟦ Expr2 ⟧)
Rule       eval⟦ Expr1 '/' Expr2 ⟧ =
             checked integer-divide(int eval⟦ Expr1 ⟧, int eval⟦ Expr2 ⟧)
Rule       eval⟦ Expr1 '*' Expr2 ⟧ =
             integer-multiply(int eval⟦ Expr1 ⟧, int eval⟦ Expr2 ⟧)
Rule       eval⟦ Expr1 '-' Expr2 ⟧ =
             integer-subtract(int eval⟦ Expr1 ⟧, int eval⟦ Expr2 ⟧)
Rule       eval⟦ Expr1 '==' Expr2 ⟧ =
             is-equal(eval⟦ Expr1 ⟧, eval⟦ Expr2 ⟧)
Rule       eval⟦ Expr1 '<=' Expr2 ⟧ =
             is-less-or-equal(int eval⟦ Expr1 ⟧, int eval⟦ Expr2 ⟧)
Rule       eval⟦ Expr1 '<' Expr2 ⟧ =
             is-less(int eval⟦ Expr1 ⟧, int eval⟦ Expr2 ⟧)
Rule       eval⟦ Expr1 '!=' Expr2 ⟧ =
             not is-equal(eval⟦ Expr1 ⟧, eval⟦ Expr2 ⟧)
Rule       eval⟦ Expr1 '>=' Expr2 ⟧ =
             is-greater-or-equal(int eval⟦ Expr1 ⟧, int eval⟦ Expr2 ⟧)
Rule       eval⟦ Expr1 '>' Expr2 ⟧ =
             is-greater(int eval⟦ Expr1 ⟧, int eval⟦ Expr2 ⟧)
Rule       eval⟦ Expr1 '&&' Expr2 ⟧ =
             if-true-else(bool eval⟦ Expr1 ⟧, bool eval⟦ Expr2 ⟧, false)
Rule       eval⟦ Expr1 '||' Expr2 ⟧ =
             if-true-else(bool eval⟦ Expr1 ⟧, true, bool eval⟦ Expr2 ⟧)
Rule       eval⟦ '!' Expr ⟧ = not(bool eval⟦ Expr ⟧)
Rule       eval⟦ 'new' '(' ')' ⟧ =
             object(fresh-atom, "Object", initialise-local-variables)
Rule       eval⟦ 'println' '(' Expr ')' ⟧ = print-line sl-to-string eval⟦ Expr ⟧
Rule       eval⟦ 'readln' '(' ')' ⟧ = [read-line]
Rule       eval⟦ 'defineFunction' '(' Expr ')' ⟧ = fail
Rule       eval⟦ 'nanoTime' '(' ')' ⟧ = fail
Rule       eval⟦ 'stacktrace' '(' ')' ⟧ = fail
Otherwise  eval⟦ Id '(' ExprList? ')' ⟧ =
             apply(fun global-bound eval⟦ Id ⟧, eval-list⟦ ExprList? ⟧)
Rule       eval⟦ Id ⟧ =
             else(assigned local-variable id⟦ Id ⟧, str id⟦ Id ⟧)
Rule       eval⟦ Id '=' Expr ⟧ =
             give(
               eval⟦ Expr ⟧,
               sequential(
                 local-variable-assign(id⟦ Id ⟧, given),
                 given))
Rule       eval⟦ Expr '.' Id ⟧ =
             scope-closed(
               object-feature-map obj eval⟦ Expr ⟧,
               else(assigned local-variable id⟦ Id ⟧, null-value))
Rule       eval⟦ Expr1 '.' Id '=' Expr2 ⟧ =
             give(
```

Syntax *ExprList* : expr-list ::= expr (‘,’ expr-list)?

Semantics eval-list⟦ _ : expr-list? ⟧ : ⇒ lists(sl-values)

Rule eval-list⟦ ⟧ = nil

Rule eval-list⟦ *Expr* ⟧ = cons(eval⟦ *Expr* ⟧, nil)

Rule eval-list⟦ *Expr* ‘,’ *ExprList* ⟧ =
cons(eval⟦ *Expr* ⟧, eval-list⟦ *ExprList* ⟧)