# Unstable-Languages-beta: SIMPLE-THR-3-Statements

The PLanCompS Project

`Unstable-Languages-beta/SIMPLE-THR/SIMPLE-THR-3-Statements/SIMPLE-THR-3-Statem`

*Language* "`SIMPLE-THR`"

## 3 Statements

*Syntax* *Block* : block ::= `{` stmts? `}`

*Stmts* : stmts ::= stmt stmts?

*Stmt* : stmt ::= imp-stmt
| vars-decl

*ImpStmt* : imp-stmt ::= block
| exp `;`
| `if (` exp `)` block (`else` block)?
| `while (` exp `)` block
| `for (` stmt exp `;` exp `)` block
| `print (` exps `) ;`
| `return` exp? `;`
| `try` block `catch (` id `)` block
| `throw` exp `;`
| `join` exp `;`
| `acquire` exp `;`
| `release` exp `;`
| `rendezvous` exp `;`

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

*Rule* $[\![$ `if (` *Exp* `)` *Block* $]\!]$ : stmt =
$[\![$ `if (` *Exp* `)` *Block* `else { }` $]\!]$

*Rule* $[\![$ `for (` *Stmt* $Exp_1$ `;` $Exp_2$ `) {` *Stmts* `}` $]\!]$ : stmt =
$[\![$ `{` *Stmt* `while (` $Exp_1$ `) { {` *Stmts* `}` $Exp_2$ `; } }` $]\!]$

```
Semantics exec⟦ _ : stmts ⟧ : ⇒ null-type
    Rule exec⟦ { } ⟧ =
           null
    Rule exec⟦ { Stmts } ⟧ =
          exec⟦ Stmts ⟧
    Rule exec⟦ ImpStmt Stmts ⟧ =
           sequential(exec⟦ ImpStmt ⟧,
             exec⟦ Stmts ⟧)
    Rule exec⟦ VarsDecl Stmts ⟧ =
           scope(declare⟦ VarsDecl ⟧,
             exec⟦ Stmts ⟧)
    Rule exec⟦ VarsDecl ⟧ =
           effect(declare⟦ VarsDecl ⟧)
    Rule exec⟦ Exp ; ⟧ =
           effect(rval⟦ Exp ⟧)
    Rule exec⟦ if ( Exp ) Block1 else Block2 ⟧ =
           if-else(rval⟦ Exp ⟧,
             exec⟦ Block1 ⟧,
             exec⟦ Block2 ⟧)
    Rule exec⟦ while ( Exp ) Block ⟧ =
           while(rval⟦ Exp ⟧,
             exec⟦ Block ⟧)
    Rule exec⟦ print ( Exps ) ; ⟧ =
           print(rvals⟦ Exps ⟧)
    Rule exec⟦ return Exp ; ⟧ =
           return(rval⟦ Exp ⟧)
    Rule exec⟦ return ; ⟧ =
           return(null)
    Rule exec⟦ try Block1 catch ( Id ) Block2 ⟧ =
           handle-thrown(exec⟦ Block1 ⟧,
             scope(bind(id⟦ Id ⟧,
                    allocate-initialised-variable(values,
                      given)),
               exec⟦ Block2 ⟧))
    Rule exec⟦ throw Exp ; ⟧ =
           throw(rval⟦ Exp ⟧)
```

SIMPLE uses natural numbers to identify threads; the use of lookup-index(_) below converts a natural number to the associated thread-id.

*Rule* exec⟦ `join` *Exp* `;` ⟧ =
    thread-join lookup-index(rval⟦ *Exp* ⟧)

The use of memo-value($V$, $SY$) below associates $V$ with a lock. When a thread requests a lock already held by another thread, the requesting thread is suspended until the request is granted. The use of postpone(_) below automatically releases held locks when the current thread terminates.

*Rule* exec⟦ `acquire` *Exp* `;` ⟧ =
    give(memo-value(rval⟦ *Exp* ⟧,
        reentrant-lock-create),
      sequential(postpone if-true-else(is-exclusive-lock-holder given,
          reentrant-lock-release given,
          null-value),
        reentrant-lock-sync-else-wait given))

The use of memo-value-recall($V$) below gives the lock associated with $V$.

*Rule* exec⟦ `release` *Exp* `;` ⟧ =
    reentrant-lock-exit memo-value-recall rval⟦ *Exp* ⟧

The use of memo-value($V$, $SY$) below associates $V$ with a rendezvous. When a thread requests a rendezvous on a particular value, and there is no previous uncompleted request for a rendezvous on the same value, the requesting thread is suspended until the request is granted.

*Rule* exec⟦ `rendezvous` *Exp* `;` ⟧ =
    rendezvous-sync-else-wait(memo-value("`rendezvous`",
        rendezvous-create(2)),
      rval⟦ *Exp* ⟧)