# The IMP Programming Language

Christiano Braga
cbraga@ic.uff.br

Instituto de Computação
Universidade Federal Fluminense

**Abstract.** IMP is a simple imperative programming language created to illustrate the use of the Basic Programming Languages Constructs library (http://github.com/ChristianoBraga/BPLCBPLC). It is implemented in the Maude language.

## 1 IMP syntax

## 2 IMP semantics

### 2.1 Genralized SMC Machines

Genralized SMC Machine Descriptions

### 2.2 GSMC Rules for IMP

## 3 Maude implementations

### 3.1 Parser

### 3.2 Compiling to BPLC

#### BPLC signature

#### Semantic functions

⟨*imp-grammar*⟩≡

```
fmod TOKEN is
    pr QID-LIST .
    sorts Token Bubble TokenList NeTokenList .
    subsort Token < TokenList .

    op _,_ : TokenList TokenList -> TokenList [assoc prec 5] .
```

```
    op token : Qid -> Token
       [special
         (id-hook Bubble        (1 1)
          op-hook qidSymbol     (<Qids> : ~> Qid)
              id-hook Exclude       ( true false nop print ))] .

    op bubble : QidList -> Bubble
       [special
         (id-hook Bubble        (1 -1)
          op-hook qidListSymbol (__ : QidList QidList ~> QidList)
          op-hook qidSymbol     (<Qids> : ~> Qid)
              id-hook Exclude       ( | if then else end { } while ; := = nop )) ] .

    op neTokenList : QidList -> NeTokenList
       [special
         (id-hook Bubble        (1 -1)
          op-hook qidListSymbol (__ : QidList QidList ~> QidList)
          op-hook qidSymbol     (<Qids> : ~> Qid)
          id-hook Exclude       ( . ))] .
endfm

-- Constants are typed at the metalevel: op c : -> S becomes the metaterm 'c.S.
-- In order to avoid including IMP-GRAMMAR into COMPILE-IMP-TO-BPLC-EXP, we
-- "cast out", from IMP-GRAMMAR, PREDICATE-DECL and COMMAND-DECL sorts and constants.
fmod PREDICATE-DECL is
        sort PredicateDecl .
    ops true false : -> PredicateDecl .
endfm

fmod COMMAND-DECL is
        sort CommandDecl .
        op nop : -> CommandDecl .
endfm

fmod IMP-GRAMMAR is
    pr TOKEN .
    pr RAT .
    inc PREDICATE-DECL .
          inc COMMAND-DECL .
    sorts VariablesDecl ConstantsDecl OperationsDecl ProcDeclList
        ProcDecl FormalsDecl BlockCommandDecl ExpressionDecl
        InitDecl InitDeclList InitDecls ClausesDecl
        ModuleDecl Expression .
```

```
subsort InitDecl < InitDeclList .
subsort VariablesDecl ConstantsDecl ProcDeclList InitDecls < ClausesDecl .
      subsort BlockCommandDecl < CommandDecl .
subsort ProcDecl < ProcDeclList .
subsort PredicateDecl < ExpressionDecl .

-- IMP BOOLEAN EXPRESSIONS
op _==_ : Token Token -> PredicateDecl [prec 30] .
op _==_ : Token ExpressionDecl -> PredicateDecl [prec 30] .
op _==_ : ExpressionDecl Token -> PredicateDecl [prec 30] .
op _==_ : ExpressionDecl ExpressionDecl -> PredicateDecl [prec 30] .
op _/\_ : PredicateDecl PredicateDecl -> PredicateDecl [assoc comm prec 35] .
op ~ : PredicateDecl -> PredicateDecl .

    -- IMP ARITHMETIC EXPRESSIONS
op _+_ : Token Token -> ExpressionDecl [gather(e E) prec 15] .
op _+_ : Token ExpressionDecl -> ExpressionDecl [gather(e E) prec 15] .
op _+_ : ExpressionDecl Token -> ExpressionDecl [gather(e E) prec 15] .
op _+_ : ExpressionDecl ExpressionDecl -> ExpressionDecl [gather(e E) prec 15] .

op _-_ : Token Token -> ExpressionDecl  [gather(e E) prec 15] .
op _-_ : Token ExpressionDecl -> ExpressionDecl [gather(e E) prec 15] .
op _-_ : ExpressionDecl Token -> ExpressionDecl [gather(e E) prec 15] .
op _-_ : ExpressionDecl ExpressionDecl -> ExpressionDecl [gather(e E) prec 15] .

op _*_ : Token Token -> ExpressionDecl [gather(e E) prec 10] .
op _*_ : Token ExpressionDecl -> ExpressionDecl [gather(e E) prec 10] .
op _*_ : ExpressionDecl Token -> ExpressionDecl [gather(e E) prec 10] .
op _*_ : ExpressionDecl ExpressionDecl -> ExpressionDecl [gather(e E) prec 10] .

op _/_ : Token Token -> ExpressionDecl [prec 20] .
op _/_ : Token ExpressionDecl -> ExpressionDecl [prec 20] .
op _/_ : ExpressionDecl Token -> ExpressionDecl [prec 20] .
op _/_ : ExpressionDecl ExpressionDecl -> ExpressionDecl [prec 20] .

-- IMP COMMANDS
op _:=_ : Token Token -> CommandDecl [prec 40] .
op _:=_ : Token ExpressionDecl -> CommandDecl [prec 40] .

op _() : Token -> CommandDecl .
op _(_) : Token Bubble -> CommandDecl .
op print : Token -> CommandDecl .
op print : ExpressionDecl -> CommandDecl .
op _;_ : CommandDecl CommandDecl -> CommandDecl [assoc prec 50] .
op _|_ : CommandDecl CommandDecl -> CommandDecl [assoc prec 50] .
```

```
    op while_do_ : PredicateDecl BlockCommandDecl -> CommandDecl [prec 40] .
    op {_} : CommandDecl -> BlockCommandDecl [prec 35] .
    op if__else_ : PredicateDecl CommandDecl CommandDecl -> CommandDecl [prec 40] .

    -- IMP DECLARATIONS
        op module__end : Token ClausesDecl -> ModuleDecl [prec 80] .
    op __ : ClausesDecl ClausesDecl -> ClausesDecl [assoc comm prec 70] .
    op var_ : TokenList -> VariablesDecl [prec 60] .
    op const_ : TokenList -> ConstantsDecl [prec 60] .
    op init_ : InitDecl -> InitDecls [prec 60] .
    op init_ : InitDeclList -> InitDecls  [prec 60] .
    op _=_ : Token Token -> InitDecl  [prec 40] .
    op _=_ : Token ExpressionDecl -> InitDecl  [prec 40] .
    op _,_ : InitDeclList InitDeclList -> InitDeclList  [assoc prec 50] .
    op proc__ : Token BlockCommandDecl -> ProcDecl [prec 50] .
    op proc_`(_`)_ : Token TokenList BlockCommandDecl -> ProcDecl  [prec 50] .
    op __ : ProcDeclList ProcDeclList -> ProcDeclList  [assoc comm prec 60] .
endfm
```

⟨*compiler-to-bplc*⟩≡

```
load ../../maude/bplc

mod COMPILE-IMP-TO-BPLC is
        inc PREDICATE-DECL .
        inc COMMAND-DECL .
    pr BPLC .
        pr META-LEVEL .

        -- Compiling IMP expressions to BPLC Exp.

    op compileId : Qid -> Id .
        eq compileId(I:Qid) = idn(downTerm(I:Qid, 'Qid)) .

    op compileId : Term -> Id .
    eq compileId('token[I:Qid]) =
       if (metaParse(upModule('RAT, false), downTerm(I:Qid, 'Qid), 'Rat)  :: ResultPair)
           then rat(downTerm(getTerm(metaParse(upModule('RAT, false),
                    downTerm(I:Qid, 'Qid), 'Rat)), 1/2))
           else
             if (metaParse(upModule('BOOL, false), downTerm(I:Qid, 'Qid), 'Bool) :: Result
             then boo(downTerm(getTerm(metaParse(upModule('BOOL, false),
                      downTerm(I:Qid, 'Qid), 'Bool)), true))
             else idn(downTerm(I:Qid, 'Qid))
             fi
           fi .

    op compileExp : Term -> Exp .

    ceq compileExp(I:Qid) = compileId(I:Qid) if not(I:Qid :: Constant) .

    eq compileExp('token[I:Qid]) = compileId('token[I:Qid]) .

    eq compileExp('_+_[T1:Term, T2:Term]) =
       add(compileExp(T1:Term), compileExp(T2:Term)) .

    eq compileExp('_-_[T1:Term, T2:Term]) =
       sub(compileExp(T1:Term), compileExp(T2:Term)) .

    eq compileExp('_*_[T1:Term, T2:Term]) =
       mul(compileExp(T1:Term), compileExp(T2:Term)) .

    eq compileExp('_/_[T1:Term, T2:Term]) =
       div(compileExp(T1:Term), compileExp(T2:Term)) .
```

```
eq compileExp('true.PredicateDecl) = boo(true) .
eq compileExp('false.PredicateDecl) = boo(false) .
eq compileExp('~[T:Term]) = neg(compileExp(T:Term)) .
eq compileExp('_/\_[T1:Term,T2:Term]) = and(compileExp(T1:Term), compileExp(T2:Term))
eq compileExp('_\/_[T1:Term,T2:Term]) = or(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_==_[T1:Term,T2:Term]) = eq(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_>_[T1:Term,T2:Term]) = gt(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_>=_[T1:Term,T2:Term]) = ge(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_<_[T1:Term,T2:Term]) = lt(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_<=_[T1:Term,T2:Term]) = le(compileExp(T1:Term), compileExp(T2:Term)) .

    -- Compiling IMP command to BPLC Cmd.

op compileCmd : Term -> Cmd .

eq compileCmd('nop.CommandDecl) = nop .

eq compileCmd('_:=_['token[I:Qid], T:Term]) =
       assign(compileId('token[I:Qid]), compileExp(T:Term)) .

eq compileCmd(''{_'}[T:Term]) = blk(compileCmd(T:Term)) .

eq compileCmd('_;_[T1:Term, T2:Term]) =
       seq(compileCmd(T1:Term), compileCmd(T2:Term)) .

eq compileCmd('_|_[T1:Term, T2:Term]) =
       choice(compileCmd(T1:Term), compileCmd(T2:Term)) .

eq compileCmd('if__else_[T1:Term, T2:Term, T3:Term]) =
       if(compileExp(T1:Term), compileCmd(T2:Term), compileCmd(T3:Term)) .

eq compileCmd('while_do_[T1:Term, T2:Term]) =
       loop(compileExp(T1:Term), compileCmd(T2:Term)) .

eq compileCmd('print[T:Term]) = print(compileId(T:Term)) .

-- Procedure calls deserve more attention...

eq compileCmd('_'(')['token[I:Qid]]) =
   cal(compileId('token[I:Qid])) .

eq compileCmd('_'(_')['token[I:Qid], 'bubble[Q:Qid]]) =
   cal(compileId('token[I:Qid]), compileActuals(Q:Qid)) .

eq compileCmd('_'(_')['token[I:Qid], 'bubble['__[TL:TermList]]]) =
```

```
       cal(compileId('token[I:Qid]), compileActuals(TL:TermList)) .

op compileActuals : TermList -> Actuals .
eq compileActuals(Q:Qid) = compileId('token[Q:Qid]) .
eq compileActuals((TL1:TermList, ''‘,.Qid , TL2:TermList)) =
       act(makeExp(TL1:TermList) , compileActuals(TL2:TermList)) .
eq compileActuals(TL:TermList) = makeExp(TL:TermList) [owise] .

op makeExp : TermList -> Exp .
eq makeExp(Q:Qid) = compileId(Q:Qid) .

ceq makeExp(TL:TermList) =
        compileExp(
             getTerm(
              metaParse(upModule('IMP-GRAMMAR, false),
               makeExprDeclQidList(TL:TermList), 'ExpressionDecl)))
if (metaParse(upModule('IMP-GRAMMAR, false),
         makeExprDeclQidList(TL:TermList), 'ExpressionDecl) ::
        ResultPair) .

op makeExprDeclQidList : TermList -> QidList .

eq makeExprDeclQidList((empty).TermList) = (nil).QidList .

ceq makeExprDeclQidList(Q:Qid) = downTerm(Q:Qid, 'error)
if downTerm(Q:Qid, 'error) =/= 'error .

eq makeExprDeclQidList((Q:Qid , TL:TermList)) =
       makeExprDeclQidList(Q:Qid) makeExprDeclQidList(TL:TermList) .

    -- Compiling IMP declarations to BPLC Dec.

-- Compiling variables. The initialization clause is necessary for properly
    -- declare variables according to BPLC ref construct.

    op compileVar : Term Term -> Dec .
    op $compileVar : Term Term -> Dec .

eq compileVar('var_[TL1:TermList], 'init_[TL2:TermList]) =
       $compileVar(TL1:TermList, TL2:TermList) .

eq $compileVar('token[I:Qid], '_=_['token[I:Qid], T:Term]) =
       ref(compileId('token[I:Qid]), compileExp(T:Term)) .

eq $compileVar('token[I:Qid], ('_‘,_['_=_['token[I:Qid], T:Term], TL:TermList])) =
```

```
   ref(compileId('token[I:Qid]), compileExp(T:Term)) .

ceq $compileVar('token[I1:Qid], ('_‘,_['_=_['token[I2:Qid], T:Term], TL:TermList])) =
    $compileVar('token[I1:Qid], TL:TermList)
    if I1:Qid =/= I2:Qid .

eq $compileVar('_‘,_['token[I:Qid], IS:TermList], '_=_['token[I:Qid], T:Term]) =
       ref(compileId('token[I:Qid]), compileExp(T:Term)) .

eq $compileVar('_‘,_['token[I:Qid], IS:TermList], '_‘,_[TL:TermList]) =
       dec($compileVar('token[I:Qid], '_‘,_[TL:TermList]),
           $compileVar(IS:TermList, '_‘,_[TL:TermList])) .

    -- Compiling constants.
    op compileConst : Term Term -> Dec .
    op $compileConst : Term Term -> Dec .

eq compileConst('const_[T1:Term], 'init_[T2:Term]) =
       $compileConst(T1:Term, T2:Term) .

eq $compileConst('token[I:Qid], '_=_['token[I:Qid], T:Term]) =
       cns(compileId('token[I:Qid]), compileExp(T:Term)) .

eq $compileConst('token[I:Qid], ('_‘,_['_=_['token[I:Qid], T:Term], TL:TermList])) =
   cns(compileId('token[I:Qid]), compileExp(T:Term)) .

ceq $compileConst('token[I1:Qid], ('_‘,_['_=_['token[I2:Qid], T:Term], TL:TermList]))
    $compileConst('token[I1:Qid], TL:TermList)
    if I1:Qid =/= I2:Qid .

eq $compileConst('_‘,_['token[I:Qid], IS:TermList], '_=_['token[I:Qid], T:Term]) =
       cns(compileId('token[I:Qid]), compileExp(T:Term)) .

eq $compileConst('_‘,_['token[I:Qid], IS:TermList], '_‘,_[TL:TermList]) =
       dec($compileConst('token[I:Qid], '_‘,_[TL:TermList]),
           $compileConst(IS:TermList, '_‘,_[TL:TermList])) .

    -- Compiling procedure declarations.
op compileProc : TermList -> Dec .
    op compileToFormals : TermList -> Formals .

    eq compileProc('__[TL1:TermList, TL2:TermList]) =
       dec(compileProc(TL1:TermList), compileProc(TL2:TermList)) .

    eq compileProc('proc__['token[O:Qid], TL:TermList, T:Term]) =
```

```
        prc(compileId('token[O:Qid]), compileCmd(T:Term)) .

    eq compileProc('proc_'(_')_['token[O:Qid], TL:TermList, T:Term]) =
        prc(compileId('token[O:Qid]),
        compileToFormals(TL:TermList), compileCmd(T:Term)) .

eq compileToFormals('token[O:Qid]) = par(compileId('token[O:Qid])) .
eq compileToFormals('_',_[TL1:TermList, TL2:TermList]) =
       for(compileToFormals(TL1:TermList), compileToFormals(TL2:TermList)) .

-- Compiling modules.
    op compileMod : Term -> Dec .
    eq compileMod('module__end['token[I:Qid],
                  '__['var_[T1:Term],
                                  '__['const_[T2:Term],
                                  '__['init_[T3:Term],
                                  T:Term]]]]) =
       dec(compileVar('var_[T1:Term],'init_[T3:Term]),
        dec(compileConst('const_[T2:Term], 'init_[T3:Term]),
             compileProc(T:Term))) .

    eq compileMod('module__end['token[I:Qid],
                  '__['var_[T1:Term],
                                  '__['init_[T3:Term],
                                  T:Term]]]) =
       dec(compileVar('var_[T1:Term],'init_[T3:Term]),
             compileProc(T:Term)) .

    eq compileMod('module__end['token[I:Qid],
                  '__['var_[T1:Term],
                                  '__['const_[T2:Term],
                                  '__['init_[T3:Term],
                                  '__[TL:TermList]]]]]) =
       dec(compileVar('var_[T1:Term],'init_[T3:Term]),
        dec(compileConst('const_[T2:Term], 'init_[T3:Term]),
             compileProc('__[TL:TermList]))) .

    eq compileMod('module__end['token[I:Qid],
                          '__['var_[T1:Term],
                                          '__['init_[T3:Term],
                                          '__[TL:TermList]]]]) =
       dec(compileVar('var_[T1:Term],'init_[T3:Term]),
             compileProc('__[TL:TermList])) .

endm
```

⟨*imp-pretty-priting*⟩≡

```
mod IMP-PRETTY-PRINTING is
    pr BPLC-MODEL-CHECKER .
       pr INT .

    -- Some auxiliary constants and functions
    ops level incr : -> Nat .
    eq level = 3 .
    eq incr = 3 .

    op printSpaces : Nat -> QidList .
    eq printSpaces(0) = (nil).QidList .
    eq printSpaces(N:Nat) = '\s printSpaces(sd(N:Nat, 1)) .

    op printTermList : TermList -> QidList .
    eq printTermList(empty) = (nil).QidList .
    eq printTermList((Q:Qid,TL:TermList)) =
       downTerm(Q:Qid,'Qid) printTermList(TL:TermList) .

    op printTokens : Term -> QidList .
       eq printTokens(Q:Qid) = Q:Qid .
       eq printTokens('token[Q:Qid]) = downTerm(Q:Qid, 'Qid) .
       eq printTokens('_`,_[T1:Term, T2:Term]) =
          printTokens(T1:Term) printToken('`,) '\s printTokens(T2:Term) .

    -- Pretty-printing IMP Expressions
    op printExp : Term -> QidList .
    eq printExp('true.PredicateDecl) = 'true .

    eq printExp('false.PredicateDecl) = 'false .

    eq printExp('bubble['__[TL:TermList]]) = printTermList(TL:TermList) .

    eq printExp('token[Q:Qid]) = downTerm(Q:Qid, 'Qid) .

    eq printExp('_==_[T1:Term, T2:Term]) =
       printExp(T1:Term) printToken('==) printExp(T2:Term) .

    eq printExp('_\/_[T1:Term, T2:Term]) =
       printExp(T1:Term) printToken('\/) printExp(T2:Term) .

    eq printExp('_/\_[T1:Term, T2:Term]) =
       printExp(T1:Term) printToken('\/) printExp(T2:Term) .

    eq printExp('~[T:Term]) =
```

```
   printToken('~) printExp(T:Term) .

eq printExp('_<_[T1:Term, T2:Term]) =
   printExp(T1:Term) printToken('<) printExp(T2:Term) .

eq printExp('_<=_[T1:Term, T2:Term]) =
   printExp(T1:Term) printToken('<=) printExp(T2:Term) .

eq printExp('_>_[T1:Term, T2:Term]) =
   printExp(T1:Term) printToken('>) printExp(T2:Term) .

eq printExp('_>=_[T1:Term, T2:Term]) =
   printExp(T1:Term) printToken('>=) printExp(T2:Term) .

eq printExp('_+_[T1:Term, T2:Term]) =
   printExp(T1:Term) printToken('+) printExp(T2:Term) .

eq printExp('_*_[T1:Term, T2:Term]) =
   printExp(T1:Term) printToken('*) printExp(T2:Term) .

eq printExp('_-_[T1:Term, T2:Term]) =
   printExp(T1:Term) printToken('-) printExp(T2:Term) .

eq printExp('_/_[T1:Term, T2:Term]) =
   printExp(T1:Term) printToken('/) printExp(T2:Term) .

eq printExp(T:Term) =
   metaPrettyPrint(upModule('IMP-GRAMMAR, false), T:Term) [owise] .

-- Pretty-printing IMP Commands
op printCmd : Term Nat -> QidList .
eq printCmd('nop.CommandDecl, N:Nat) = 'nop .

eq printCmd('print[T:Term], N:Nat) =
   printToken('print) printToken('`() printExp(T:Term) printToken('`)) .

eq printCmd('_`(_`)[T1:Term, T2:Term], N:Nat) =
   printExp(T1:Term) printToken('`() printExp(T2:Term) printToken('`)) .

eq printCmd('_|_[T1:Term, '_;_[T2:TermList]], N:Nat) =
   printSpaces(N:Nat)
   printCmd(T1:Term, N:Nat) printToken('|) '\n
   printSpaces(N:Nat) printToken('`() printCmd('_;_[T2:TermList], N:Nat) printToken('`

ceq printCmd('_|_[T:TermList, F:Qid[TL:TermList]], N:Nat) =
```

```
    printSpaces(N:Nat) printCmd(T:TermList, N:Nat) printToken('|)
    printCmd(F:Qid[TL:TermList], N:Nat)
 if F:Qid =/= '_;_ .

ceq printCmd('_;_[F1:Qid[TL1:TermList], F2:Qid[TL2:TermList]], N:Nat) =
    printCmd(F1:Qid[TL1:TermList], N:Nat) '\s printToken(';)
    printCmd(F2:Qid[TL2:TermList], N:Nat)
 if F1:Qid =/= '_;_ and F2:Qid =/= '_;_ .

ceq printCmd('_;_[F1:Qid[TL1:TermList], F2:Qid[TL2:TermList]], N:Nat) =
    printSpaces(N:Nat) printCmd(F1:Qid[TL1:TermList], N:Nat) '\s printToken(';) '\n
    printSpaces(N:Nat) printCmd(F1:Qid[TL1:TermList], N:Nat)
 if F1:Qid == '_;_ or F2:Qid == '_;_ .

eq printCmd('_:=_[T1:Term, T2:Term], N:Nat) =
   printExp(T1:Term) printToken(':=) printExp(T2:Term) .

eq printCmd('while_do_[T1:Term, T2:Term], N:Nat) =
   printToken('while) ' printToken('`() printExp(T1:Term) printToken('`)) '
   printToken('do)
   printCmd(T2:Term, N:Nat) .

eq printCmd('if__else_[T1:Term, T2:Term, T3:Term], N:Nat) =
   printToken('if) ' printToken('`() printExp(T1:Term) printToken('`)) '
       printCmd(T2:Term, N:Nat)
   '\s printToken('else) printCmd(T3:Term, N:Nat) .

eq printCmd('`{_`}[C:Constant], N:Nat) =
   '\s printToken('`{) printCmd(C:Constant, 0) printToken('`}) .

eq printCmd('`{_`}['_;_[TL:TermList]], N:Nat) =
   '\s printToken('`{) '\n
   printSpaces(N:Nat + incr) printCmd('_;_[TL:TermList], N:Nat + incr) '\n
   printSpaces(N:Nat) printToken('`}) .

eq printCmd('`{_`}['_|_[TL:TermList]], N:Nat) =
   '\s printToken('`{) '\n printCmd('_|_[TL:TermList], N:Nat + incr) '\n
   printSpaces(N:Nat) printToken('`}) .

ceq printCmd('`{_`}[F:Qid[TL:TermList]], N:Nat) =
   '\s printToken('`{) '\n
   printSpaces(N:Nat + incr) printCmd(F:Qid[TL:TermList], N:Nat + incr) '\n
   printSpaces(N:Nat) printToken('`})
if F:Qid =/= '_;_ /\ F:Qid =/= '_|_ .
```

```
eq printCmd(T:Term, N:Nat) =
   printSpaces(N:Nat) metaPrettyPrint(upModule('IMP-GRAMMAR, false), T:Term) [owise] .

-- Pretty-printing IMP Declarations
op printModule : Term -> QidList .
eq printModule('module__end[T1:Term, T2:Term]) =
       printToken('module) printTokens(T1:Term)
       printClauses(T2:Term)
       '\n printToken('end) .

op printClauses : Term -> QidList .
eq printClauses('proc__[T1:Term, T2:Term]) =
       '\n printSpaces(level)
   printToken('proc) printTokens(T1:Term)
   printCmd(T2:Term, level) .

eq printClauses('__['proc__[T1:Term, T2:Term], T3:Term]) =
       '\n printSpaces(level) printToken('proc) printTokens(T1:Term)
   printCmd(T2:Term, level)
   printClauses(T3:Term) .

eq printClauses('proc_`(_`)_[T1:Term, T2:Term, T3:Term]) =
       '\n printSpaces(level)
   printToken('proc) printTokens(T1:Term)
       printToken('`() printTokens(T2:Term) printToken('`))
   printCmd(T3:Term, level) .

eq printClauses('__['proc_`(_`)_[T1:Term, T2:Term, T3:Term], T4:Term]) =
       '\n printSpaces(level) printToken('proc) printTokens(T1:Term) printToken('`()
   printTokens(T2:Term) printToken('`)) printCmd(T3:Term, level)
   printClauses(T4:Term) .

eq printClauses('__['init_[T1:Term], T2:Term]) =
       '\n printSpaces(level) printToken('init) printInit(T1:Term) printClauses(T2:Ter

eq printClauses('__['const_[T1:Term], T2:Term]) =
       '\n printSpaces(level) printToken('const) printTokens(T1:Term) printClauses(T2:

eq printClauses('__['var_[T1:Term], T2:Term]) =
       '\n printSpaces(level) printToken('var) printTokens(T1:Term) printClauses(T2:Te

op printInit : Term -> QidList .
    eq printInit('_=_[T1:Term, T2:Term]) =
       printTokens(T1:Term) printToken('=) printExp(T2:Term) .
eq printInit('_`,_[T1:Term, T2:Term]) =
```

```
       printInit(T1:Term) printToken(’‘,) ’\s printInit(T2:Term) .

-- Pretty-printing parse error
op printQidList : QidList -> QidList .
eq printQidList(nil) = (nil).QidList .
eq printQidList(Q:Qid QL:QidList) = printToken(Q:Qid) printQidList(QL:QidList) .

op printToken : Qid -> Qid .
eq printToken(Q:Qid) = ’\b ’\! Q:Qid ’\o .

op printInputWithError : QidList Nat -> QidList .
op $printInputWithError : QidList Int QidList -> QidList .
eq printInputWithError(QL:QidList, N:Nat) =
       $printInputWithError(QL:QidList, N:Nat, (nil).QidList) .

eq $printInputWithError(nil, I:Int, QL:QidList) = QL:QidList .

eq $printInputWithError(QL:QidList, 0, (nil).QidList) = (nil).QidList .

ceq $printInputWithError(QL1:QidList, 0, (QL2:QidList Q:Qid)) =
    QL2:QidList ’\r ’\u ’»> Q:Qid ’«< ’HERE ’\o
        printQidList(QL1:QidList)
if QL2:QidList =/= nil .

eq $printInputWithError(Q:Qid QL1:QidList, I:Int, QL2:QidList) =
       $printInputWithError(QL1:QidList, (I:Int + (- 1)), QL2:QidList
       printToken(Q:Qid)) .

op printParseError : QidList ResultPair? -> Qid .
eq printParseError(’module Q:Qid QL:QidList , noParse(N:Nat)) =
       ’IMP: ’Error ’at ’position
       metaPrettyPrint(upModule(’NAT, false), upTerm(N:Nat))
       ’while ’parsing ’Module Q:Qid ’: ’\n
       printInputWithError((’module Q:Qid QL:QidList), N:Nat) .

eq printParseError(QL:QidList, ambiguity(T1:ResultPair, T2:ResultPair)) =
       ’IMP: ’\r ’Ambiguous ’parse ’\o ’\n
       metaPrettyPrint(upModule(’IMP-GRAMMAR, false), getTerm(T1:ResultPair)) ’\n
       ’\r ’vs. ’\o ’\n
       metaPrettyPrint(upModule(’IMP-GRAMMAR, false), getTerm(T2:ResultPair)) .

-- Pretty-print exec and mc output
op printState : Env Store -> QidList .
eq printState(noEnv, S:Store) = (nil).QidList .
```

```
eq printState((I:Id |-> bind(L:Loc)) , ((L:Loc |-> store(R:Rat)), S:Store)) =
       printId(I:Id) '= metaPrettyPrint(upModule('RAT, false), upTerm(R:Rat)) .

ceq printState(((I:Id |-> bind(L:Loc)), E:Env) ,
        ((L:Loc |-> store(R:Rat)), S:Store)) =
        printId(I:Id) '=
        metaPrettyPrint(upModule('RAT, false), upTerm(R:Rat))
        printState(E:Env , (L:Loc |-> store(R:Rat), S:Store))
    if E:Env =/= noEnv .

eq printState((I:Id |-> bind(L:Loc)), (L:Loc |-> store(B:Bool), S:Store)) =
       printId(I:Id) '= metaPrettyPrint(upModule('BOOL, false), upTerm(B:Bool)) .

ceq printState(((I:Id |-> bind(L:Loc)), E:Env) ,
               ((L:Loc |-> store(B:Bool)), S:Store)) =
        printId(I:Id) '= metaPrettyPrint(upModule('RAT, false), upTerm(B:Bool)) '`,
        printState(E:Env , (L:Loc |-> store(B:Bool), S:Store))
if E:Env =/= noEnv .

eq printState(((I:Id |-> B:Bindable), E:Env) , S:Store) =
   printState(E:Env , S:Store) [owise] .

op printId : Id -> Qid .
eq printId(idn(Q:Qid)) = (Q:Qid).Qid .

op printCnt : ControlStack ValueStack -> QidList .
eq printCnt((LOOP C:ControlStack), (V:Value val(loop(E:Exp, K:Cmd)) VS:ValueStack)) =
       printToken('while) printToken('`() printExp(E:Exp) printToken('`)) '... .

eq printCnt((choice(K1:Cmd, K2:Cmd) C:ControlStack), VS:ValueStack) =
       printCmd(K1:Cmd) printToken('|) printCmd(K2:Cmd) .

eq printCnt(C:ControlStack, VS:ValueStack) =
       '\n 'Constrol 'stack:
           metaPrettyPrint(upModule('BPLC+META-LEVEL, false),
           upTerm(C:ControlStack))
       '\n 'Value 'stack:
       metaPrettyPrint(upModule('BPLC+META-LEVEL, false),
       upTerm(VS:ValueStack)) .

op printExp : Exp -> QidList .

eq printExp(rat(R:Rat)) = '\g metaPrettyPrint(upModule('RAT, false),
   upTerm(R:Rat)) '\o .
```

```
eq printExp(boo(B:Bool)) = '\g metaPrettyPrint(upModule('BOOL, false),
   upTerm(B:Bool)) '\o .

eq printExp(neg(E:Exp)) = printToken('~) printExp(E:Exp) .
eq printExp(and(E1:Exp, E2:Exp)) = printExp(E1:Exp) printToken('/\) printExp(E2:Exp) .
eq printExp(eq(E1:Exp, E2:Exp)) = printExp(E1:Exp) printToken('==) printExp(E2:Exp) .
eq printExp(idn(Q:Qid)) = printTokens(Q:Qid) .
eq printExp(E:Exp) = metaPrettyPrint(upModule('BPLC, false), upTerm(E:Exp)) .

op printCmd : Cmd -> QidList .
eq printCmd(seq(C1:Cmd, C2:Cmd)) = printCmd(C1:Cmd) printToken(';) printCmd(C2:Cmd) .
eq printCmd(assign(I:Id, E:Exp)) = printExp(I:Id) printToken(':=) printExp(E:Exp) .
eq printCmd(if(E:Exp, C1:Cmd, C2:Cmd)) = printToken('if) printExp(E:Exp) '... .

eq printCmd(choice(C1:Cmd, C2:Cmd)) = printCmd(C1:Cmd)
   printToken('|) printCmd(C2:Cmd) .

eq printCmd(C:Cmd) = metaPrettyPrint(upModule('BPLC, false), upTerm(C:Cmd)) .

op printConf : Conf -> QidList .

eq printConf(< env : E:Env , sto : S:Store , exc : X:Exc , cnt :
      C:ControlStack, val : V:ValueStack, ...:Set{SemComp} > ) =
  if X:Exc == CNT
      then printCnt(C:ControlStack, V:ValueStack) '`[ printState(E:Env, S:Store) '`]
      else 'IMP: 'Internal 'error 'while 'printing 'configuration.  fi .

op printValueStack : ValueStack -> QidList .
eq printValueStack(evs) = '\b 'No 'output. '\o .

eq printValueStack(val(R:Rat) evs) =
   metaPrettyPrint(upModule('RAT, false), upTerm(R:Rat)) .

eq printValueStack(val(B:Bool) evs) =
   metaPrettyPrint(upModule('BOOL, false), upTerm(B:Bool)) .

eq printValueStack(V:Value VS:ValueStack) =
       printValueStack(V:Value) printValueStack(VS:ValueStack) .

op printOut : Conf -> QidList .
eq printOut(< out : O:ValueStack , ...:Set{SemComp} > ) =
   printValueStack(O:ValueStack) .

op printExec : Term ~> QidList .
ceq printExec(T:Term) = printOut(downTerm(T:Term, < noSemComp >))
```

```
    if downTerm(T:Term, < noSemComp >) =/= < noSemComp > .

    op printTraceStep : TraceStep -> QidList .
    eq printTraceStep({T:Term, Y:Type, R:Rule}) =
       printConf(downTerm(T:Term, < noSemComp >)) .

    op printTrace : Trace Nat -> QidList .
    eq printTrace(nil, N:Nat) = (nil).QidList .
    eq printTrace(TS:TraceStep, N:Nat) =
           '\b 'State metaPrettyPrint(upModule('NAT, false), upTerm(N:Nat)) ': '\o '\n
           printTraceStep(TS:TraceStep) .

    ceq printTrace(TS:TraceStep T:Trace, N:Nat) =
            '\b 'State metaPrettyPrint(upModule('NAT, false), upTerm(N:Nat)) ': '\o '\n
            printTraceStep(TS:TraceStep) '\n printTrace(T:Trace, (N:Nat + 1))
    if T:Trace =/= nil .

    op printTransitionList : TransitionList -> QidList .
    eq printTransitionList(nil) = (nil).QidList .
    eq printTransitionList({C:Conf, R:RuleName}) = printConf(C:Conf) .
    ceq printTransitionList({C:Conf, R:RuleName} TL:TransitionList) =
            printConf(C:Conf) '\b '-> '\o printTransitionList(TL:TransitionList)
    if TL:TransitionList =/= nil .

    op printModelCheckResult : ModelCheckResult QidList -> QidList .
    eq printModelCheckResult(B:Bool, QL:QidList) =
           'IMP: '\b 'Model 'check 'result 'to 'command '\c QL:QidList '\o '\b ': '\o '\n
           metaPrettyPrint(upModule('BOOL, false), upTerm(B:Bool)) .

    eq printModelCheckResult(counterexample(TL1:TransitionList, TL2:TransitionList), QL:Qi
           'IMP: '\b 'Model 'check 'counter 'example 'to 'command '\c QL:QidList '\o '\b
           '\b 'Path 'from 'the 'initial 'state: '\o
           '\n printTransitionList(TL1:TransitionList)
           '\n '\b 'Loop: '\o
           '\n printTransitionList(TL2:TransitionList) .
endm
```

⟨*imp-interface*⟩≡

```
mod IMP-INTERFACE is
    pr LOOP-MODE * (sort State to LoopState).
        pr COMPILE-IMP-TO-BPLC .
        pr IMP-PRETTY-PRINTING .

    sorts Dec? MetaIMPModule Command IMPState .
        subsort Term < MetaIMPModule .
        subsort Dec < Dec? .
        subsort IMPState < LoopState .

    op noDec : -> Dec? .
        op noModule : -> MetaIMPModule .
        op idle : -> Command .
        op <_;_;_> : MetaIMPModule Dec? QidList -> IMPState .

        op init : -> System .
    op init‘IMP‘no‘banner : -> System .
    op banner : -> QidList .
    eq banner = ’\g ’IMP ’Prototype ’\o ’ ’‘( ’\y ’\! ’March ’2018 ’\o ’‘) .
    eq init = [nil, < noModule ; noDec ; nil >, banner] .
    eq init‘IMP‘no‘banner = [nil, < noModule ; noDec ; nil >, nil] .

    vars QIL QIL’ QIL’’ QIL1 QIL2 : QidList .

    rl [version] : [’version, L:LoopState, QIL] => [nil, L:LoopState, banner] .

    -- Loading a module.
    crl [in] : [’module Q:Qid QIL, < M:MetaIMPModule ; D:Dec? ; QIL’ >, QIL’’] =>
        if (T:ResultPair? :: ResultPair)
        then [nil, < getTerm(T:ResultPair?) ;
                                        compileMod(getTerm(T:ResultPair?)) ;
                                        ’IMP: ’\b ’Module Q:Qid ’loaded. ’\o  >, QIL’’]
        else [nil, < noModule ; noDec ; nil >,
                  printParseError(’module Q:Qid QIL, T:ResultPair?)]
        fi
    if T:ResultPair? :=
           metaParse(upModule(’IMP-GRAMMAR, false), ’module Q:Qid QIL, ’ModuleDecl) .

        -- Viewing a module.
    crl [view] : [’view , < M:MetaIMPModule ; D:Dec ; QIL’ >, QIL’’] =>
                    [nil, < M:MetaIMPModule ; D:Dec ; QIL’ >,
                                    printModule(M:MetaIMPModule)]
    if M:MetaIMPModule =/= noModule .
```

```
-- Executing a command.
crl [exec-unbounded] : [('exec QIL), < M:MetaIMPModule ; D:Dec ; QIL' >, QIL''] =>
    if P:ResultPair? :: ResultPair
    then [nil, < M:MetaIMPModule ; D:Dec ; QIL' > ,
        if compileCmd(getTerm(P:ResultPair?)) :: Cmd
        then
                    'IMP: '\b 'Execution 'result 'for '\c QIL '\o '\b ': '\o
                    printExec(getTerm(metaRewrite(upModule('BPLC+META-LEVEL, false),
                     'run[upTerm(compileCmd(getTerm(P:ResultPair?))), upTerm(D:Dec)]
        else 'IMP: '\r 'Internal 'Error 'while 'compiling '\s QIL
        fi ]
    else [nil, < M:MetaIMPModule ; D:Dec ; QIL' >,
              'IMP: '\r 'Error 'while 'processing 'command '\o 'exec '\s QIL ]
    fi
if P:ResultPair? :=
       metaParse(upModule('IMP-GRAMMAR, false), QIL, 'CommandDecl) .

-- Model check command
crl [mc] : [('mc QIL1 '|= QIL2), < 'module__end[T1:Term, T2:Term] ; D:Dec ; QIL' >, QI
    if ((P1:ResultPair? :: ResultPair) and (P2:ResultPair? :: ResultPair))
    then [nil, < 'module__end[T1:Term, T2:Term] ; D:Dec ; QIL' >,
          if compileCmd(getTerm(P1:ResultPair?)) :: Cmd
              then
           printModelCheckResult(downTerm(
               getTerm(metaReduce(
                 makeMCModule(printTokens(T1:Term), D:Dec),
                 '_`,_|=?_[upTerm(D:Dec),
                           upTerm(compileCmd(getTerm(P1:ResultPair?))),
                           getTerm(P2:ResultPair?)])),true), QIL1 '\s '|= '\s QIL2)
          else 'IMP: '\r 'Internal 'Error 'while 'processing 'command '\o 'mc QIL1 '|=
          fi ]
    else [nil, < 'module__end[T1:Term, T2:Term] ; D:Dec ; QIL' >,
              'IMP: '\r 'Error 'while 'processing 'command '\o 'mc QIL1 '|= QIL2]
    fi
if P1:ResultPair? := metaParse(upModule('IMP-GRAMMAR, false), QIL1, 'CommandDecl) /\
       P2:ResultPair? := metaParse(makeMCModule(printTokens(T1:Term), D:Dec), QIL2, 'P


    -- Printing
crl [out] : [nil, < M:MetaIMPModule ; D:Dec? ; QIL' > , QIL''] =>
             [nil, < M:MetaIMPModule ; D:Dec? ; nil > , QIL']
if QIL' =/= nil .

-- Command error
crl [com-error] : [Q:Qid QIL, < M:MetaIMPModule ; D:Dec ; QIL' > , QIL''] =>
```

```
        [nil, < M:MetaIMPModule ; D:Dec ; QIL' > ,
             'IMP: '\r 'Error 'no 'such 'command '\o Q:Qid QIL]
    if (Q:Qid =/= 'module) /\ (Q:Qid =/= 'view) /\
           (Q:Qid =/= 'exec) /\ (Q:Qid =/= 'mc) .
endm

loop init .
```