Ebnf group *decls*

File ::= [*DeclList*]
DeclList ::= { *OneDecl* }
OneDecl ::= *LetDecl*
| *ExternDecl*
| *ExceptDecl*
| *NodeDecl*
ExceptDecl ::= **exception** *IdentList*
LetDecl ::= **let <ident>** *OptParams OptType* **=** *Statement*
ExternDecl ::= **extern <ident>** *OptParams OptType*
NodeStart ::= **node** | **system**
NodeDecl ::= *NodeStart* **<ident> (** *TypedIdentListOpt* **) returns (** *TypedIdentList* **) =** *Statement*

Ebnf group *varparams*

IdentList ::= *Ident* { **,** *Ident* }
TypedIdent ::= *IdentList* **:** *Type* [**=** *Exp*]
TypedIdentListOpt ::= [*TypedIdentList*]
TypedIdentList ::= *TypedIdentListA* [**;**]
TypedIdentListA ::= *TypedIdent* { **;** *TypedIdent* }
OptParams ::= [**(** [*TypedParamList*] **)**]
TypedParamList ::= *TypedParam* { **;** *TypedParam* }
TypedParam ::= *IdentList* **:** *ParamType*
OptType ::= [**:** *Type*]

Ebnf group *types*

Type ::= *PredefType*
| **trace**
PredefType ::= **bool**
| **int**
| **real**
ParamType ::= *Type*
| *PredefType* **ref**

Ebnf group *statements*

Statement ::= *Exp*
| **raise** *Ident*
| *Statement* **fby** *Statement*
| *LoopStatement*
| *BracedStatement*

```
                    |  LetDecl in Statement
                    |  assert Exp in Statement
                    |  exist TypedIdentList in Statement
                    |  exception IdentList in Statement
                    |  try Statement DoPart
                    |  catch Ident in Statement DoPart
                    |  trap Ident in Statement DoPart
DoPart          ::= [ do Statement ]
LoopStatement   ::= loop [ Average | Gaussian ] Statement
Average         ::= [ Exp , Exp ]
Gaussian        ::= ~ Exp [ : Exp ]
WeightOpt       ::= [ weight Exp ]
ChoicesList     ::= Statement WeightOpt { | Statement WeightOpt }
PrioList        ::= Statement { |> Statement }
ParaList        ::= Statement { &> Statement }
BracedStatement ::= { Statement [ |> PrioList | WeightOpt | ChoicesList | &> ParaList ]
                    }
```

Ebnf group *expressions*

```
Exp         ::= IdentRef
              |  Constant
              |  pre Ident
              |  ( Exp )
              |  - Exp
              |  not Exp
              |  BinExp
              |  if Exp then Exp else Exp
BinExp      ::= Exp ( = | <> | or | xor | and | => | + | - | * | / | div | mod | < | <= | > | >= )
                Exp
Constant    ::= true
              |  false
              |  <integer>
              |  <floating>
```

Ebnf group *identref*

```
IdentRef    ::= Ident [ ( [ ArgList ] ) ]
ArgList     ::= Arg { , Arg }
Arg         ::= Statement
```