```
grammar HOStipula ;

@lexer::members {

 * PARSER RULES

prog :  STIPULA contract_id = ID CLPAR (assetdecl)? (fielddecl)? INIT init_state = ID agreement fun+ CRPAR ;

agreement : (AGREEMENT LPAR party (COMMA party)* RPAR LPAR vardec (COMMA vardec)* RPAR CLPAR (assign)+ CRPAR IMPL
AT state);

assetdecl :  ASSET idAsset+=ID (',' idAsset+=ID)* ;

fielddecl : FIELD idField+=ID (',' idField+=ID)* ;

fun :  ((AT state)* party (COMMA party)* COLON funId=ID LPAR (vardec ( COMMA vardec)* )? RPAR SLPAR (assetdec
( COMMA assetdec)* )? SRPAR ( body | hobody) ) ;

body : (LPAR prec RPAR)? CLPAR (stat)+ SEMIC (events)+ CRPAR IMPL AT state ;

hobody  :  HOLPAR HID HORPAR   ;

hocode : ('parties' party (COMMA party)*)? (assetdecl)? (fielddecl)? fun* CLPAR (stat)+ SEMIC (events)+ CRPAR IMPL
AT state   ;

assign : (party (COMMA party)* COLON vardec (COMMA vardec)*)  ;

dec : (ASSET | FIELD) ID  ;

type :  INTEGER | DOUBLE | BOOLEAN | STRING ;

state : ID;
```

```
party : ID;

vardec : ID ;

assetdec : ID ;

varasm : vardec ASM expr ;

stat     :       EMPTY
            |   left=value operator=ASSETUP right=ID (COMMA rightPlus=ID)?
            |   left=value operator=FIELDUP right=(ID | EMPTY)
            | ifelse

            ;

ifelse : (IF LPAR cond=expr RPAR CLPAR ifBranch+=stat (ifBranch+=stat)* CRPAR (ELSEIF condElseIf+=expr CLPAR
elseIfBranch+=stat (elseIfBranch+=stat)* CRPAR)* (ELSE CLPAR elseBranch+=stat (elseBranch+=stat)* CRPAR )?);

events  :      EMPTY
            | ( expr TRIGGER AT ID CLPAR stat+ CRPAR IMPL AT ID )
            ;

prec   : expr
       ;

expr     :  ('-')? left=term (operator=(PLUS | MINUS | OR) right=expr)?
       ;

term    : left=factor (operator=(TIMES | DIV  | AND) right=term)?
       ;
```

```
factor : left=value (operator = (EQ | LE | GE | LEQ | GEQ | NEQ ) right=value)?
      ;

value  :  number
      | ID
      | NOW
      | LPAR expr RPAR
      | RAWSTRING
      | EMPTY
      | (TRUE | FALSE)
      ;

real : number DOT number ;

number : INT | REAL ;


 * LEXER RULES
SEMIC    : ';' ;
COLON    : ':' ;
COMMA    : ',' ;
DOT      : '.' ;
EQ       : '==' ;
NEQ      : '!=' ;
IMPL     : '==>' ;
ASM      : '=' ;
ASSETUP :  '-o' ;
FIELDUP : '->' ;
PLUS     : '+' ;
MINUS    : '-' ;
TIMES    : '*' ;
DIV      : '/' ;
```

```
AT      : '@' ;
TRUE    : 'true' ;
FALSE   : 'false' ;
LPAR    : '(' ;
RPAR    : ')' ;
SLPAR   : '[' ;
SRPAR   : ']' ;
HOLPAR  : '([' ;
HORPAR  : '])' ;
CLPAR   : '{' ;
CRPAR   : '}' ;
LEQ     : '<=';
GEQ     : '>=';
LE      : '<';
GE      : '>';
OR      : '||';
AND     : '&&';
NOT     : '!';
EMPTY   : '_' ;
NOW     : 'now' ;
TRIGGER : '>>';
IF     : 'if' ;
ELSEIF   : 'else if' ;
ELSE   : 'else' ;
STIPULA : 'stipula';
ASSET : 'asset' ;
FIELD : 'field' ;
AGREEMENT : 'agreement';
INTEGER : 'int' ;
DOUBLE : 'real' ;
BOOLEAN : 'bool' ;
STRING : 'string' ;
```

```
PARTY : 'party' ;
INIT : 'init' ;

RAWSTRING : '\'' ~('\'')+ '\'' | '"' ~('"')+ '"' ;

INT : '0' | [1-9] [0-9]* ;

REAL : [0-9]* '.' [0-9]+ ;

WS
 : [ \t\r\n] -> skip
 ;

//IDs
fragment CHAR  : 'a'..'z' |'A'..'Z' ;
ID : CHAR (CHAR | INT | EMPTY)* ;
HID: 'A'..'Z' ;

OTHER
 : .
 ;

//ESCAPED SEQUENCES
LINECOMENTS    : '//' (~('\n'|'\r'))* -> skip;
BLOCKCOMENTS    : '/*'( ~('/'|'*')|'/'~'*'|'*'~'/'|BLOCKCOMENTS)* '*/' -> skip;

 //VERY SIMPLISTIC ERROR CHECK FOR THE LEXING PROCESS, THE OUTPUT GOES DIRECTLY TO THE TERMINAL
 //THIS IS WRONG!!!!
ERR     : . { System.out.println("Invalid char: "+ getText()); lexicalErrors++; } -> channel(HIDDEN);
```