AtomC - generarea de cod

```
fnDef: ( typeBase | VOID ) ID
    LPAR ( fnParam ( COMMA fnParam )* )? RPAR
    {
    addInstr(&fn->fn.instr,OP_ENTER);
    }
    stmCompound[false]
    {
    fn->fn.instr->arg.i=symbolsLen(fn->fn.locals);
    if(fn->type.tb==TB_VOID)
        addInstrWithInt(&fn->fn.instr,OP_RET_VOID,symbolsLen(fn->fn.params));
    /* dropDomain(); */
    }
```

```
stm: stmCompound
           | IF LPAR expr RPAR
                {
                addRVal(&owner->fn.instr,rCond.lval,&rCond.type);
                Type intType={TB_INT,NULL,-1};
                insertConvIfNeeded(lastInstr(owner->fn.instr),&rCond.type,&intType);
                Instr *ifJF=addInstr(&owner->fn.instr,OP_JF);
                }
                stm ( ELSE
                    {
                    Instr *ifJMP=addInstr(&owner->fn.instr,OP_JMP);
                    ifJF->arg.instr=addInstr(&owner->fn.instr,OP_NOP);
                    }
                    stm {ifJMP->arg.instr=addInstr(&owner->fn.instr,OP_NOP);} |
                    {
                    ifJF->arg.instr=addInstr(&owner->fn.instr,OP_NOP);
                    } )
           | WHILE {Instr *beforeWhileCond=lastInstr(owner->fn.instr);} LPAR expr RPAR
                {
                addRVal(&owner->fn.instr,rCond.lval,&rCond.type);
                Type intType={TB_INT,NULL,-1};
                insertConvIfNeeded(lastInstr(owner->fn.instr),&rCond.type,&intType);
                Instr *whileJF=addInstr(&owner->fn.instr,OP_JF);
                }
                stm
                {
                addInstr(&owner->fn.instr,OP_JMP)->arg.instr=beforeWhileCond->next;
                whileJF->arg.instr=addInstr(&owner->fn.instr,OP_NOP);
                }
           | RETURN ( expr
                {
                addRVal(&owner->fn.instr,rExpr.lval,&rExpr.type);
                insertConvIfNeeded(lastInstr(owner->fn.instr),&rExpr.type,&owner->type);
                addInstrWithInt(&owner->fn.instr,OP_RET,symbolsLen(owner->fn.params));
                } | {addInstr(&owner->fn.instr,OP_RET_VOID);} ) SEMICOLON
           | (expr {if(rExpr.type.tb!=TB_VOID)addInstr(&owner->fn.instr,OP_DROP);} )?
```

```
SEMICOLON
```

```
exprAssign: exprUnary ASSIGN exprAssign
    {
    addRVal(&owner->fn.instr,r->lval,&r->type);
    insertConvIfNeeded(lastInstr(owner->fn.instr),&r->type,&rDst.type);
    switch(rDst.type.tb){
        case TB_INT:addInstr(&owner->fn.instr,OP_STORE_I);break;
        case TB_DOUBLE:addInstr(&owner->fn.instr,OP_STORE_F);break;
        }
    }
    | exprOr
```

```
exprRel: {Token *op;} exprRel ( LESS[op] | LESSEQ[op] | GREATER[op] | GREATEREQ[op] )
    {
    Instr *lastLeft=lastInstr(owner->fn.instr);
    addRVal(&owner->fn.instr,r->lval,&r->type);
    }
    exprAdd
    {
    addRVal(&owner->fn.instr,right.lval,&right.type);
    insertConvIfNeeded(lastLeft,&r->type,&tDst);
    insertConvIfNeeded(lastInstr(owner->fn.instr),&right.type,&tDst);
    switch(op->code){
        case LESS:
            switch(tDst.tb){
                case TB_INT:addInstr(&owner->fn.instr,OP_LESS_I);break;
                case TB_DOUBLE:addInstr(&owner->fn.instr,OP_LESS_F);break;
                }
            break;
        }
    /* *r=(Ret){{TB_INT,NULL,-1},false,true}; */
    }
    | exprAdd
```

```
exprAdd: {Token *op;} exprAdd ( ADD[op] | SUB[op] )
    {
    Instr *lastLeft=lastInstr(owner->fn.instr);
    addRVal(&owner->fn.instr,r->lval,&r->type);
    }
    exprMul
    {
    addRVal(&owner->fn.instr,right.lval,&right.type);
    insertConvIfNeeded(lastLeft,&r->type,&tDst);
    insertConvIfNeeded(lastInstr(owner->fn.instr),&right.type,&tDst);
    switch(op->code){
        case ADD:
            switch(tDst.tb){
                case TB_INT:addInstr(&owner->fn.instr,OP_ADD_I);break;
                case TB_DOUBLE:addInstr(&owner->fn.instr,OP_ADD_F);break;
            }
            break;
        case SUB:
            switch(tDst.tb){
                case TB_INT:addInstr(&owner->fn.instr,OP_SUB_I);break;
                case TB_DOUBLE:addInstr(&owner->fn.instr,OP_SUB_F);break;
                }
            break;
        }
    /* *r=(Ret){tDst,false,true}; */
    }
    | exprMul
```

```
exprMul: {Token *op;} exprMul ( MUL[op] | DIV[op] )
  {
  Instr *lastLeft=lastInstr(owner->fn.instr);
  addRVal(&owner->fn.instr,r->lval,&r->type);
  }
  exprCast
  {
  addRVal(&owner->fn.instr,right.lval,&right.type);
  insertConvIfNeeded(lastLeft,&r->type,&tDst);
  insertConvIfNeeded(lastInstr(owner->fn.instr),&right.type,&tDst);
  switch(op->code){
    case MUL:
      switch(tDst.tb){
        case TB_INT:addInstr(&owner->fn.instr,OP_MUL_I);break;
        case TB_DOUBLE:addInstr(&owner->fn.instr,OP_MUL_F);break;
        }
      break;
    case DIV:
      switch(tDst.tb){
        case TB_INT:addInstr(&owner->fn.instr,OP_DIV_I);break;
        case TB_DOUBLE:addInstr(&owner->fn.instr,OP_DIV_F);break;
        }
    break;
  }
  /* *r=(Ret){tDst,false,true}; */
  }
  | exprCast
```

```
exprPrimary: ID[tkName] ( LPAR ( expr[&rArg]
  {
  addRVal(&owner->fn.instr,rArg.lval,&rArg.type);
  insertConvIfNeeded(lastInstr(owner->fn.instr),&rArg.type,&param->type);
  /*param=param->next;*/
  }
    ( COMMA expr[&rArg]
      {
      addRVal(&owner->fn.instr,rArg.lval,&rArg.type);
      insertConvIfNeeded(lastInstr(owner->fn.instr),&rArg.type,&param->type);
      /*param=param->next;*/
      }
      )* )? RPAR
  {
  if(s->fn.extFnPtr){
    addInstr(&owner->fn.instr,OP_CALL_EXT)->arg.extFnPtr=s->fn.extFnPtr;
    }else{
    addInstr(&owner->fn.instr,OP_CALL)->arg.instr=s->fn.instr;
    }
  }
  |
  {
  if(s->kind==SK_VAR){
    if(s->owner==NULL){    // global variables
      addInstr(&owner->fn.instr,OP_ADDR)->arg.p=s->varMem;
      }else{    // local variables
      switch(s->type.tb){
        case TB_INT:addInstrWithInt(&owner->fn.instr,OP_FPADDR_I,s->varIdx+1);break;
        case TB_DOUBLE:addInstrWithInt(&owner->fn.instr,OP_FPADDR_F,s->varIdx+1);break;
        }
      }
    }
  if(s->kind==SK_PARAM){
    switch(s->type.tb){
      case TB_INT:
addInstrWithInt(&owner->fn.instr,OP_FPADDR_I,s->paramIdx-symbolsLen(s->owner->fn.params)-
1); break;
      case TB_DOUBLE:
addInstrWithInt(&owner->fn.instr,OP_FPADDR_F,s->paramIdx-symbolsLen(s->owner->fn.params)-
1); break;
      }
    }
  }
  )
    | INT[&ct]    {addInstrWithInt(&owner->fn.instr,OP_PUSH_I,ct->i);}
    | DOUBLE[&ct]    {addInstrWithDouble(&owner->fn.instr,OP_PUSH_F,ct->d);}
    | CHAR[&ct]
    | STRING[&ct]
    | LPAR expr[r] RPAR
```