

AtomC - analiza de tipuri

```
// IF - conditia trebuie sa fie scalar
// WHILE - conditia trebuie sa fie scalar
// RETURN - expresia trebuie sa fie scalar
// RETURN - functiile void nu pot returna o valoare
// RETURN - functiile non-void trebuie sa aiba o expresie returnata, a carei tip sa fie
convertibil la tipul returnat de functie
stm: {Ret rCond,rExpr;} stmCompound[true]
    | IF LPAR expr[&rCond]
        {if(!canBeScalar(&rCond))tkerr("the if condition must be a scalar value");}
        RPAR stm ( ELSE stm )?
    | WHILE LPAR expr[&rCond]
        {if(!canBeScalar(&rCond))tkerr("the while condition must be a scalar value");}
        RPAR stm
    | RETURN ( expr[&rExpr]
        {
            if(owner->type.tb==TB_VOID)tkerr("a void function cannot return a value");
            if(!canBeScalar(&rExpr))tkerr("the return value must be a scalar value");
            if(!convTo(&rExpr.type,&owner->type))tkerr("cannot convert the return expression
type to the function return type");
        }
        |
        {if(owner->type.tb!=TB_VOID)tkerr("a non-void function must return a value");}
        ) SEMICOLON
    | expr[&rExpr]? SEMICOLON
```

```
expr[out Ret *r]: exprAssign[r]
```

```
// Destinatia trebuie sa fie left-value
// Destinatia nu trebuie sa fie constanta
// Ambii operanzi trebuie sa fie scalari
// Sursa trebuie sa fie convertibila la destinatie
// Tipul rezultat este tipul sursei
exprAssign[out Ret *r]: {Ret rDst;}exprUnary[&rDst] ASSIGN exprAssign[r]
{
    if(!rDst.lval)tkerr("the assign destination must be a left-value");
    if(rDst.ct)tkerr("the assign destination cannot be constant");
    if(!canBeScalar(&rDst))tkerr("the assign destination must be scalar");
    if(!canBeScalar(r))tkerr("the assign source must be scalar");
    if(!convTo(&r->type,&rDst.type))tkerr("the assign source cannot be converted to
destination");
    r->lval=false;
    r->ct=true;
}
| exprOr[r]
```

```
// Ambii operanzi trebuie sa fie scalari si sa nu fie structuri
// Rezultatul este un int
exprOr[out Ret *r]: exprOr[r] OR {Ret right;} exprAnd[&right]
```

```
{
Type tDst;
if(!arithTypeTo(&r->type,&right.type,&tDst))tkerr("invalid operand type for ||");
*r=(Ret){{TB_INT,NULL,-1},false,true};
}
| exprAnd[r]
```

```
// Ambii operanzi trebuie sa fie scalari si sa nu fie structuri
// Rezultatul este un int
exprAnd[out Ret *r]: exprAnd[r] AND {Ret right;} exprEq[&right]
{
Type tDst;
if(!arithTypeTo(&r->type,&right.type,&tDst))tkerr("invalid operand type for &&");
*r=(Ret){{TB_INT,NULL,-1},false,true};
}
| exprEq[r]
```

```
// Ambii operanzi trebuie sa fie scalari si sa nu fie structuri
// Rezultatul este un int
exprEq[out Ret *r]: exprEq[r] ( EQUAL | NOTEQ ) {Ret right;} exprRel[&right]
{
Type tDst;
if(!arithTypeTo(&r->type,&right.type,&tDst))tkerr("invalid operand type for == or
!=");
*r=(Ret){{TB_INT,NULL,-1},false,true};
}
| exprRel[r]
```

```
// Ambii operanzi trebuie sa fie scalari si sa nu fie structuri
// Rezultatul este un int
exprRel[out Ret *r]: exprRel[r] ( LESS | LESSEQ | GREATER | GREATEREQ ) {Ret right;}
exprAdd[&right]
{
Type tDst;
if(!arithTypeTo(&r->type,&right.type,&tDst))tkerr("invalid operand type for <, <=, >,
>=");
*r=(Ret){{TB_INT,NULL,-1},false,true};
}
| exprAdd[r]
```

```
// Ambii operanzi trebuie sa fie scalari si sa nu fie structuri
exprAdd[out Ret *r]: exprAdd[r] ( ADD | SUB ) {Ret right;} exprMul[&right]
{
Type tDst;
if(!arithTypeTo(&r->type,&right.type,&tDst))tkerr("invalid operand type for + or -");
*r=(Ret){tDst,false,true};
}
| exprMul[r]
```

```
// Ambii operanzi trebuie sa fie scalari si sa nu fie structuri
```

```

exprMul[out Ret *r]: exprMul[r] ( MUL | DIV ) {Ret right;} exprCast[&right]
{
    Type tDst;
    if(!arithTypeTo(&r->type,&right.type,&tDst))tkerr("invalid operand type for * or /");
    *r=(Ret){tDst,false,true};
}
| exprCast[r]

```

```

// Structurile nu se pot converti
// Tipul la care se converteste nu poate fi structura
// Un array se poate converti doar la alt array
// Un scalar se poate converti doar la alt scalar
exprCast[out Ret *r]: LPAR {Type t;Ret op;} typeBase[&t] arrayDecl[&t]? RPAR
exprCast[&op]
{
    if(t.tb==TB_STRUCT)tkerr("cannot convert to a struct type");
    if(op.type.tb==TB_STRUCT)tkerr("cannot convert a struct");
    if(op.type.n>=0&&t.n<0)tkerr("an array can be converted only to another array");
    if(op.type.n<0&&t.n>=0)tkerr("a scalar can be converted only to another scalar");
    *r=(Ret){t,false,true};
}
| exprUnary[r]

```

```

// Minus unar si Not trebuie sa aiba un operand scalar
// Rezultatul lui Not este un int
exprUnary[out Ret *r]: ( SUB | NOT ) exprUnary[r]
{
    if(!canBeScalar(r))tkerr("unary - or ! must have a scalar operand");
    r->lval=false;
    r->ct=true;
}
| exprPostfix[r]

```

```

// Doar un array poate fi indexat
// Indexul in array trebuie sa fie convertibil la int
// Operatorul de selectie a unui camp de structura se poate aplica doar structurilor
// Campul unei structuri trebuie sa existe
exprPostfix[out Ret *r]: exprPostfix[r] LBRACKET {Ret idx;} expr[&idx] RBRACKET
{
    if(r->type.n<0)tkerr("only an array can be indexed");
    Type tInt={TB_INT,NULL,-1};
    if(!convTo(&idx.type,&tInt))tkerr("the index is not convertible to int");
    r->type.n=-1;
    r->lval=true;
    r->ct=false;
}
| exprPostfix[r] DOT ID[tkName]
{
    if(r->type.tb!=TB_STRUCT)tkerr("a field can only be selected from a struct");
    Symbol *s=findSymbolInList(r->type.s->structMembers,tkName->text);
    if(!s)tkerr("the structure %s does not have a field
%s",r->type.s->name,tkName->text);
}

```

```

    *r=(Ret){s->type,true,s->type.n>=0};
}
| exprPrimary[r]

```

```

// ID-ul trebuie sa existe in TS
// Doar functiile pot fi apelate
// 0 functie poate fi doar apelata
// Apelul unei functii trebuie sa aiba acelasi numar de argumente ca si numarul de
parametri de la definitia ei
// Tipurile argumentelor de la apelul unei functii trebuie sa fie convertibile la
tipurile parametrilor functiei
exprPrimary[out Ret *r]: ID[tkName]
{
    Symbol *s=findSymbol(tkName->text);
    if(!s)tkerr("undefined id: %s",tkName->text);
}
( LPAR
    {
        if(s->kind!=SK_FN)tkerr("only a function can be called");
        Ret rArg;
        Symbol *param=s->fn.params;
    }
    ( expr[&rArg]
        {
            if(!param)tkerr("too many arguments in function call");
            if(!convTo(&rArg.type,&param->type))tkerr("in call, cannot convert the
argument type to the parameter type");
            param=param->next;
        }
        ( COMMA expr[&rArg]
            {
                if(!param)tkerr("too many arguments in function call");
                if(!convTo(&rArg.type,&param->type))tkerr("in call, cannot convert
the argument type to the parameter type");
                param=param->next;
            }
        )* )? RPAR
    {
        if(param)tkerr("too few arguments in function call");
        *r=(Ret){s->type,false,true};
    }
    |
    {
        if(s->kind==SK_FN)tkerr("a function can only be called");
        *r=(Ret){s->type,true,s->type.n>=0};
    }
)
| INT {*r=(Ret){{TB_INT,NULL,-1},false,true};}
| DOUBLE {*r=(Ret){{TB_DOUBLE,NULL,-1},false,true};}
| CHAR {*r=(Ret){{TB_CHAR,NULL,-1},false,true};}
| STRING {*r=(Ret){{TB_CHAR,NULL,0},false,true};}
| LPAR expr[r] RPAR

```