

# AtomC - analiza de domeniu

```
// numele structurii trebuie sa fie unic in domeniu
// in interiorul structurii nu pot exista doua variabile cu acelasi nume
structDef: STRUCT ID[tkName] LACC
{
    Symbol *s=findSymbolInDomain(symTable,tkName->text);
    if(s)tkerr("symbol redefinition: %s",tkName->text);
    s=addSymbolToDomain(symTable,newSymbol(tkName->text,SK_STRUCT));
    s->type.tb=TB_STRUCT;
    s->type.s=s;
    s->type.n=-1;
    pushDomain();
    owner=s;
}
varDef* RACC SEMICOLON
{
    owner=NULL;
    dropDomain();
}
```

```
// numele variabilei trebuie sa fie unic in domeniu
// variabilele de tip vector trebuie sa aiba dimensiunea data (nu se accepta: int v[])
varDef: {Type t;} typeBase[&t] ID[tkName]
( arrayDecl[&t]
    {if(t.n==0)tkerr("a vector variable must have a specified dimension");}
)? SEMICOLON
{
    Symbol *var=findSymbolInDomain(symTable,tkName->text);
    if(var)tkerr("symbol redefinition: %s",tkName->text);
    var=newSymbol(tkName->text,SK_VAR);
    var->type=t;
    var->owner=owner;
    addSymbolToDomain(symTable,var);
    if(owner){
        switch(owner->kind){
            case SK_FN:
                var->varIdx=symbolsLen(owner->fn.locals);
                addSymbolToList(&owner->fn.locals,dupSymbol(var));
                break;
            case SK_STRUCT:
                var->varIdx=typeSize(&owner->type);
                addSymbolToList(&owner->structMembers,dupSymbol(var));
                break;
        }
    }else{
        var->varMem=safeAlloc(typeSize(&t));
    }
}
```

```
// daca tipul de baza este o structura, ea trebuie sa fie deja definita anterior
```

```

typeBase[out Type *t]: {t->n=-1;}
(
  INT {t->tb=TB_INT;}
  | DOUBLE {t->tb=TB_DOUBLE;}
  | CHAR {t->tb=TB_CHAR;}
  | STRUCT ID[tkName]
    {
      t->tb=TB_STRUCT;
      t->s=findSymbol(tkName->text);
      if(!t->s)tkerr("structura nedefinita: %s",tkName->text);
    }
)

```

```

arrayDecl[inout Type *t]: LBRACKET
  ( CT_INT[tkSize] {t->n=tkSize->i;} | {t->n=0;} )
RBRACKET

```

```

// numele functiei trebuie sa fie unic in domeniu
// domeniul local functiei incepe imediat dupa LPAR
// corpul functiei {...} nu defineste un nou subdomeniu in domeniul local functiei
fnDef: {Type t;}
( typeBase[&t] | VOID {t.tb=TB_VOID;} ) ID[tkName] LPAR
{
  Symbol *fn=findSymbolInDomain(symTable,tkName->text);
  if(fn)tkerr("symbol redefinition: %s",tkName->text);
  fn=newSymbol(tkName->text,SK_FN);
  fn->type=t;
  addSymbolToDomain(symTable,fn);
  owner=fn;
  pushDomain();
}
( fnParam ( COMMA fnParam )* )? RPAR stmCompound[false]
{
  dropDomain();
  owner=NULL;
}

```

```

// numele parametrului trebuie sa fie unic in domeniu
// parametrii pot fi vectori cu dimensiune data, dar in acest caz li se sterge dimensiunea ( int v[10] -> int v[] )
fnParam: {Type t;} typeBase[&t] ID[tkName]
(arrayDecl[&t] {t.n=0;} )?
{
  Symbol *param=findSymbolInDomain(symTable,tkName->text);
  if(param)tkerr("symbol redefinition: %s",tkName->text);
  param=newSymbol(tkName->text,SK_PARAM);
  param->type=t;
  param->owner=owner;
  param->paramIdx=symbolsLen(owner->fn.params);
  // parametrul este adaugat atat la domeniul curent, cat si la parametrii fn
  addSymbolToDomain(symTable,param);
  addSymbolToList(&owner->fn.params,dupSymbol(param));
}

```

```
}
```

```
// corpul compus {...} al instructiunilor defineste un nou domeniu  
stm: stmCompound[true] ...
```

```
// se defineste un nou domeniu doar la cerere  
stmCompound[in bool newDomain]: LACC  
  {if(newDomain)pushDomain();}  
  ( varDef | stm )* RACC  
  {if(newDomain)dropDomain();}
```

```
// deoarece typeBase si arrayDecl au nevoie de un argument, se adauga acesta  
// t va fi folosit ulterior  
exprCast: LPAR {Type t;} typeBase[&t] arrayDecl[&t]? RPAR exprCast|exprUnary
```