Quick - analiza de tipuri

// Reguli semantice pentru analiza de tipuri:

1. simbolurile trebuie definite anterior folosirii lor
2. funcțiile se pot doar apela
3. un apel de funcție trebuie să aibă același număr de argumente cu cel de la definirea funcției
4. argumentele de la apelul funcției trebuie să aibă tipurile identice cu cele de la definire
5. o variabilă nu se poate apela ca funcție
6. tipul returnat de return trebuie să fie identic cu cel returnat de funcție
7. return poate exista doar într-o funcție
8. condițiile pentru if și while trebuie să aibă tipul int sau real
9. nu există conversii implicite între tipurile de date, nici măcar între int și real. Din acest motiv operanzii operatorilor aritmetici binari și de comparație (+ - * / < ==) trebuie să aibă tipuri identice. Operanzii operatorilor logici binari (&& ||) pot avea tipuri diferite.
10. operatorii aritmetici și logici (+ - (inclusiv unar) * / && || !) sunt definiți doar pentru tipurile int și real
11. atribuirea și operatorii de comparație (= < ==) sunt definiți pentru toate tipurile
12. operatorii de comparatie și logici (&& || ! < ==) returnează int 0 sau 1 (false/true)

program ::=
{
addPredefinedFns(); // it will be inserted after the code for domain analysis
}
(**defVar** | **defFunc** | **block**)* FINISH
defVar ::= VAR ID COLON **baseType** SEMICOLON
baseType ::= TYPE_INT | TYPE_REAL | TYPE_STR
defFunc ::= FUNCTION ID LPAR **funcParams** RPAR COLON **baseType** **defVar*** **block** END
block ::= **instr**+
funcParams ::= (**funcParam** (COMMA **funcParam**)*)?
funcParam ::= ID COLON **baseType**
instr ::= **expr**? SEMICOLON
| IF LPAR **expr**
{
if(ret.type==TYPE_STR)tkerr("the if condition must have type int or real");
}
RPAR **block** (ELSE **block**)? END
| RETURN **expr**
{
if(!crtFn)tkerr("return can be used only in a function");
if(ret.type!=crtFn->type)tkerr("the return type must be the same as the function return type");
}
SEMICOLON
| WHILE LPAR **expr**
{
if(ret.type==TYPE_STR)tkerr("the while condition must have type int or real");
}
RPAR **block** END
expr ::= **exprLogic**
exprLogic ::= **exprAssign** ((AND | OR)
{
Ret leftType=ret;
if(leftType.type==TYPE_STR)tkerr("the left operand of && or || cannot be of type str");
}
exprAssign
{
if(ret.type==TYPE_STR)tkerr("the right operand of && or || cannot be of type str");

```
        setRet(TYPE_INT,false);
        }
        )*
exprAssign ::= ID
        {
        const char *name=consumed->text;
        }
        ASSIGN exprComp
        {
        Symbol *s=searchSymbol(name);
        if(!s)tkerr("undefined symbol: %s",name);
        if(s->kind==KIND_FN)tkerr("a function (%s) cannot be used as a destination for assignment ",name);
        if(s->type!=ret.type)tkerr("the source and destination for assignment must have the same type");
        ret.lval=false;
        }
        | exprComp
exprComp ::= exprAdd ( ( LESS | EQUAL )
        {
        Ret leftType=ret;
        }
        exprAdd
        {
        if(leftType.type!=ret.type)tkerr("different types for the operands of < or ==");
        setRet(TYPE_INT,false);        // the result of comparation is int 0 or 1
        }
        )?
exprAdd ::= exprMul ( ( ADD | SUB )
        {
        Ret leftType=ret;
        if(leftType.type==TYPE_STR)tkerr("the operands of + or - cannot be of type str");
        }
        exprMul
        {
        if(leftType.type!=ret.type)tkerr("different types for the operands of + or -");
        ret.lval=false;
        }
        )*
exprMul ::= exprPrefix ( ( MUL | DIV )
        {
        Ret leftType=ret;
        if(leftType.type==TYPE_STR)tkerr("the operands of * or / cannot be of type str");
        }
        exprPrefix
        {
        if(leftType.type!=ret.type)tkerr("different types for the operands of * or /");
        ret.lval=false;
        }
        )*
exprPrefix ::= SUB factor
        {
        if(ret.type==TYPE_STR)tkerr("the expression of unary - must be of type int or real");
        ret.lval=false;
        }
        | NOT factor
        {
        if(ret.type==TYPE_STR)tkerr("the expression of ! must be of type int or real");
```

```
        setRet(TYPE_INT,false);
        }
        | factor
factor ::= INT
        {
        setRet(TYPE_INT,false);
        }
        | REAL
        {
        setRet(TYPE_REAL,false);
        }
        | STR
        {
        setRet(TYPE_STR,false);
        }
        | LPAR expr RPAR
        | ID
        {
        Symbol *s=searchSymbol(consumed->text);
        if(!s)tkerr("undefined symbol: %s",consumed->text);
        }
        ( LPAR
                {
                if(s->kind!=KIND_FN)tkerr("%s cannot be called, because it is not a function",s->name);
                Simbol *argDef=s->args;
                }
                ( expr
                        {
                        if(!argDef)tkerr("the function %s is called with too many arguments",s->name);
                        if(argDef->type!=ret.type)tkerr("the argument type at function %s call is different from the
                one given at its definition",s->name);
                        argDef=argDef->next;
                        }
                        ( COMMA expr
                                {
                                if(!argDef)tkerr("the function %s is called with too many arguments",s->name);
                                if(argDef->type!=ret.type)tkerr("the argument type at function %s call is different
                        from the one given at its definition",s->name);
                                argDef=argDef->next;
                                }
                                )* )? RPAR
                                        {
                                        if(argDef)tkerr("the function %s is called with too few arguments",s->name);
                                        setRet(s->type,false);
                                        }
                                        |
                                        {
                                        if(s->kind==KIND_FN)tkerr("the function %s can only be called",s->name);
                                        setRet(s->type,true);
                                        }
                                        )
```