

Identificação

de nomes para a linguagem Cymbol

- Capítulo 8.4 do livro ANTLR 4 Reference contém uma implementação com 2 listeners:
 - 1ª passagem usa um listener para definir símbolos: variáveis, funções
 - 2ª passagem usa outro listener para o acesso aos símbolos
- A 2ª passagem só é necessária no caso da linguagem permitir que certos símbolos possam ser referenciados antes de serem declarados
 - o que acontece com esta linguagem no que diz respeito ao nome de funções

- Alterei ligeiramente o código que está no livro, e coloquei-o na tutoria: ficheiro [cymbol-symtab.zip](#)
- As alterações feitas estão descritas em [readme.txt](#)
 - *classe Symbol inclui Token em vez de só ter o nome*
 - *argumentos de função ficam no mesmo scope que símbolos declarados localmente na função*
- Recomenda-se a leitura do capítulo 8.4 do ANTLR 4 Reference.

```
public class Symbol { // A generic programming language symbol
    public static enum Type {tINVALID, tVOID, tINT, tFLOAT}

    Token token;
    Type type;
    Scope scope;        // All symbols know what scope contains them.

    public Symbol(Token token) { this.token = token; }
    public Symbol(Token token, Type type) { this(token); this.type = type; }
    public Token getToken() { return token; }
    public String lexeme() { return getToken().getText(); }

    public String toString() {
        if ( type!=Type.tINVALID ) return '<'+lexeme()+":"+type+'>';
        return lexeme();
    }
}
```

Casos particulares de símbolos:

Variable e Function

```
public class VariableSymbol extends Symbol {  
    public VariableSymbol(Token token, Type type) { super(token, type); }  
}
```

```
public class FunctionSymbol extends Symbol {  
    ArrayList<Symbol> arguments = new ArrayList<Symbol>();  
  
    public FunctionSymbol(Token token, Type retType) { super(token, retType); }  
  
    public ArrayList<Symbol> get_arguments() { return arguments; }  
    public void add_argument( Symbol sym ) { arguments.add(sym); }  
    public String toString() { return "function"+super.toString()+":"+arguments; }  
}
```

```
public class Scope {  
    Scope enclosingScope;    // null if global (outermost) scope  
    String name;              // for debug, there's really no need for it otherwise.  
    Map<String, Symbol> symbols = new LinkedHashMap<String, Symbol>();  
  
    public Scope(Scope enclosingScope) {  
        this.enclosingScope = enclosingScope;  
        this.name = "noname";  
    }  
  
    public Scope(Scope enclosingScope, String name) {  
        this.enclosingScope = enclosingScope;  
        this.name = name;  
    }  
}
```

```
// look for identifier name in this scope alone.  
// return Symbol if found, and null otherwise.  
public Symbol resolve_local(String name) { return symbols.get(name); }  
  
public Symbol resolve(String name) {  
    Symbol s = resolve_local(name);  
    if ( s!=null ) return s;  
    // if not here, check any enclosing scope  
    if ( enclosingScope != null ) return enclosingScope.resolve(name);  
    return null; // not found  
}
```

```
public void define(Symbol sym) {  
    symbols.put(sym.lexeme(), sym);  
    sym.scope = this; // track the scope in each symbol  
}
```

Listener para definição de símbolos

```
public class DefPhase extends CymbolBaseListener {  
    ParseTreeProperty<Scope> scopes = new ParseTreeProperty<Scope>();  
    Scope global;  
    Scope currentScope;    // define symbols in this scope
```

Iremos guardar o scope em nós relevantes da árvore (i.e., cada vez que entramos num bloco)

Necessitamos disto para podermos passar a informação dos scopes ao 2º listener.

Exemplo Implementação *Listener*

```
public void enterFile(CymbolParser.FileContext ctx) {  
    global = new Scope(null, "global");  
    currentScope = global;  
    System.out.println(currentScope);  
}
```

Iremos definir os símbolos
no currentScope

Ao início cria-se o scope global,
que não tem "pai"
(enclosingScope == null)



```
public void exitVarDecl(CymbolParser.VarDeclContext ctx) {  
    defineVar(ctx.type(), ctx.ID().getSymbol());  
}
```

```
void defineVar(CymbolParser.TypeContext typeCtx, Token nameToken) {  
    int typeTokenType = typeCtx.start.getType();  
    Symbol.Type type = CheckSymbols.getType(typeTokenType);  
    VariableSymbol var = new VariableSymbol(nameToken, type);  
    if ( !currentScope.contains(var.lexeme()) )  
        currentScope.define(var); // Define symbol in current scope  
    else  
        CheckSymbols.error(nameToken, "variable " + var.lexeme()  
            + " is already defined");  
}
```



```
public class CheckSymbols {  
    public static Symbol.Type getType(int tokenType) {  
        switch ( tokenType ) {  
            case CymbolParser.K_VOID : return Symbol.Type.tVOID;  
            case CymbolParser.K_INT : return Symbol.Type.tINT;  
            case CymbolParser.K_FLOAT : return Symbol.Type.tFLOAT;  
        }  
        return Symbol.Type.tINVALID;  
    }  
  
    public static void error(Token t, String msg) {  
        System.err.printf("line %d:%d %s\n", t.getLine(),  
            t.getCharPositionInLine(), msg);  
    }  
}
```

- Ainda na fase de definição de símbolos temos de fazer algo em:

enterFunctionDecl

exitFormalParameter

enterBlock

exitBlock

- Vejamos os detalhes

```
public void enterFunctionDecl(CymbolParser.FunctionDeclContext ctx) {  
    Token token = ctx.ID().getSymbol();  
    int typeTokenType = ctx.type().start.getType();  
    Symbol.Type type = CheckSymbols.getType(typeTokenType);  
    FunctionSymbol function = new FunctionSymbol(token, type);  
    currentScope.define(function); // Define function in current scope  
    currentFunction = function;  
}
```

Não criamos um novo scope aqui. Iremos fazê-lo mais tarde

currentFunction é uma variável de instância do listener

Criamos um FunctionSymbol e definimo-lo no scope corrente.

exitFunctionDecl(...) não necessita de fazer nada.

exitFormalParameter

```
public void exitFormalParameter(CymbolParser.FormalParameterContext ctx) {  
    CymbolParser.TypeContext typeCtx = ctx.type();  
    int typeTokenType = typeCtx.start.getType();  
    Symbol.Type type = CheckSymbols.getType(typeTokenType);  
    VariableSymbol var = new VariableSymbol(ctx.ID().getSymbol(), type);  
    currentFunction.add_argument(var);  
    System.out.println("currentFunction " + currentFunction.lexeme()  
        + " : " + currentFunction);  
}
```

Criamos um VariableSymbol mas adiamos a sua inclusão num scope. Guardamo-lo apenas na lista de argumentos da função.

- Ao entrarmos num bloco, criamos um novo scope e verificamos se *currentFunction* está definido.
 - *Em caso afirmativo, definimos os argumentos (previamente guardados) no scope corrente.*
 - *Em caso negativo, significa que se trata de um bloco local*
 - *Ao final anotamos o scope no nó da árvore (saveScope)*



```
public void enterBlock(CymbolParser.BlockContext ctx) {
    // push new scope
    currentScope = new Scope(currentScope);
    if (currentFunction != null) {
        currentScope.setName( currentFunction.lexeme() );
        // add function parameters to the current scope,
        // as if they were local variables
        for (Symbol sym: currentFunction.get_arguments()) {
            if ( !currentScope.contains(sym.lexeme()) )
                currentScope.define(sym);
            else
                CheckSymbols.error(sym.getToken(), "formal parameter "
                                   + sym.lexeme()
                                   + " is defined more than once in function "
                                   + currentFunction.lexeme() );
        }
    }
    else
        currentScope.setName("local");
    currentFunction = null;
    saveScope(ctx, currentScope);
}
```




```
public void exitBlock(CymbolParser.BlockContext ctx) {  
    currentScope = currentScope.getEnclosingScope(); // pop scope  
}
```

main() cria 2 listeners

- Após a definição dos símbolos podemos criar um novo listener para o acesso/referência aos ditos símbolos
 - *que supostamente deverão ter sido declarados.*
 - *caso não estejam → mensagem de erro*

```
ParseTreeWalker walker = new ParseTreeWalker();  
DefPhase def = new DefPhase();  
walker.walk(def, tree);  
// create next phase and feed symbol table info from def to ref phase  
RefPhase ref = new RefPhase(def.global, def.scopes);  
walker.walk(ref, tree);
```



Passamos ao 2º listener o scope global e os scopes que anotamos aos nós (blocos) da árvore sintática


```
public class RefPhase extends CymbolBaseListener {
    ParseTreeProperty<Scope> scopes;
    Scope globals;
    Scope currentScope; // resolve symbols starting in this scope

    public RefPhase(Scope globals, ParseTreeProperty<Scope> scopes) {
        this.scopes = scopes;
        this.globals = globals;
    }

    public void enterFile(CymbolParser.FileContext ctx) {
        currentScope = globals;
    }
}
```

enterBlock, exitBlock


```
public void enterBlock(CymbolParser.BlockContext ctx) {  
    currentScope = scopes.get(ctx);  
}  
public void exitBlock(CymbolParser.BlockContext ctx) {  
    currentScope = currentScope.getEnclosingScope();  
}
```



Ao entramos num bloco, obtemos o scope respectivo
que gravamos durante a fase de definição dos símbolos



```
public void exitVar(CymbolParser.VarContext ctx) {  
    String name = ctx.ID().getSymbol().getText();  
    Symbol var = currentScope.resolve(name);  
    if ( var==null ) {  
        CheckSymbols.error(ctx.ID().getSymbol(), "no such variable: "+name);  
    }  
    if ( var instanceof FunctionSymbol ) {  
        CheckSymbols.error(ctx.ID().getSymbol(), name+" is not a variable");  
    }  
}
```



Se não estiver na tabela de símbolos → erro
Se estiver, mas não for uma variável → erro

```
public void exitCall(CymbolParser.CallContext ctx) {  
    String funcName = ctx.ID().getText();  
    Symbol meth = currentScope.resolve(funcName);  
    if ( meth==null ) {  
        CheckSymbols.error(ctx.ID().getSymbol(), "no such function: "+funcName);  
    }  
    if ( meth instanceof VariableSymbol ) {  
        CheckSymbols.error(ctx.ID().getSymbol(), funcName+" is not a function");  
    }  
}
```



Se não estiver na tabela de símbolos → erro
Se estiver, mas não for uma função → erro