

# 5. Tablas de símbolos para alcance monolítico y anidado

- Definición y conceptos
- Patrones

# Definición y conceptos

- Las aplicaciones basadas en lenguajes rastrean los símbolos en una estructura de datos llamada tabla de símbolos.
- Para construir una tabla de símbolos, debemos formalizar lo que hacemos de manera implícita cuando escribimos software.

```
class T { ... }; // define class T
T f() { ... }    // define function f returning type T
int x;           // define variable x of type int
```

- De manera inconsciente definimos 3 símbolos (entidades de programa) en nuestra mente: la clase T, la función (o método) f y la variable x.

# Definición y conceptos

- Para construir una aplicación lenguaje, es necesario imitar esto en software. Para las definiciones anteriores, necesitamos hacer algo así:

```
Type c = new ClassSymbol("T");                                // define class
MethodSymbol m = new MethodSymbol("f", c);                      // define method
Type intType = new BuiltInTypeSymbol("int");                   // define int type
VariableSymbol v = new VariableSymbol("x", intType);           // define var x
```

- Estos constructores, nos indican lo que debemos saber de cada símbolo.

# Definición y conceptos

- Cada uno de estos símbolos tiene al menos las siguientes propiedades: nombre, categoría y tipo.
  - **Nombre.** Estos símbolos son usualmente identificadores como `x`, `f`, `T`, pero también pueden ser operadores. En el lenguaje Ruby, podemos usar operadores como `+` para dar nombre a un método.
  - **Categoría.** Nos indica que es el símbolo, es una clase, método, variable, etiqueta, etc.
  - **Tipo.** Para validar operaciones como `x + y`, es necesario conocer los tipos de `x` e `y`.

# Definición y conceptos

- Una tabla de símbolos implementa cada categoría en una clase diferente, manteniendo el nombre y tipo como propiedades.

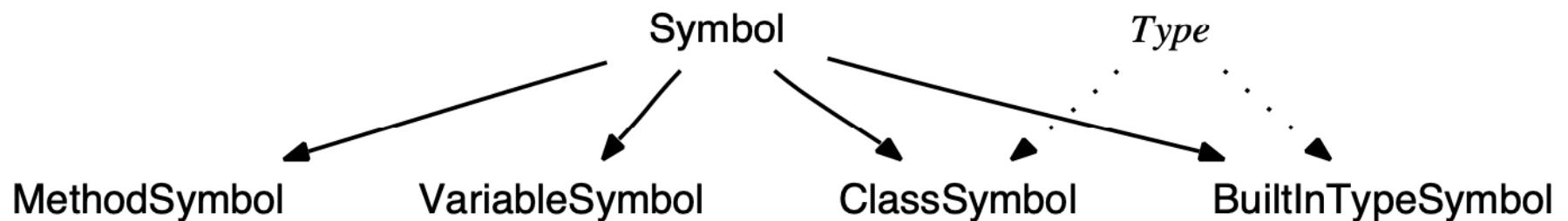
```
public class Symbol {  
    public String name; // All symbols at least have a name  
    public Type type; // Symbols have types  
}
```

- Las variables son las categorías más simples y se pueden representar de la siguiente manera:

```
public class VariableSymbol extends Symbol {  
    public VariableSymbol(String name, Type type) { super(name, type); }  
}
```

# Definición y conceptos

- Para distinguir entre los tipos definidos por el usuario y otros símbolos, es una buena idea etiquetar con una interfaz de tipo.



```
public class BuiltInTypeSymbol extends Symbol implements Type {  
    public BuiltInTypeSymbol(String name) { super(name); }  
}
```

```
public interface Type { public String getName(); }
```

# Definición y conceptos - Alcance

- El alcance (scope) es una región de código, con límites bien definidos, que agrupa la definición de símbolos en un diccionario asociado a esa región de código.
- Los límites en el alcance usualmente coinciden con tokens de inicio y fin (llaves o palabras reservadas).
- A esto lo llamamos alcance léxico porque la extensión del alcance está delimitado léxicamente.
- Un término mejor sería alcance estático porque podemos rastrear el alcance con tan sólo ver el código fuente, es decir, sin ejecutarlo.

# Definición y conceptos - Alcance

- **Alcance estático vs dinámico.** La mayoría de los lenguajes tienen alcance estático, pero algunos como Lisp o PostScript tiene alcance dinámico. El alcance dinámico permite a los métodos ver las variables locales de los métodos que se invocan.
- **Alcance con nombre.** Muchos alcances, como clases y métodos tienen nombres, pero el alcance global y local no los tienen.
- **Anidamiento.** Los lenguajes usualmente permiten algún tipo de alcance de anidamiento. Por ejemplo, los bloques de código encerrados en llaves.
- **Contenido.** Algunos alcances permiten declaraciones, algunos permiten enunciados, otros permiten ambos. Por ejemplo, las estructuras (structs) en C sólo permiten declaraciones.
- **Visibilidad.** Los símbolos dentro de un alcance pueden o no ser visibles para otra sección de código. Por ejemplo, los campos dentro de una estructura en C son visibles para cualquier sección de código. Los campos dentro de una clase tienen modificadores de visibilidad como public y private que indican cuales secciones de código pueden referenciarlas.

# Definición y conceptos - Alcance

- Para representar un alcance, utilizaremos una interfaz, de esta forma podemos etiquetar entidades como funciones y clases.
- Por ejemplo, una función es un símbolo que también juega el papel de un alcance.
- Los alcances pueden tener nombre y tienen apuntadores a sus alcances internos.

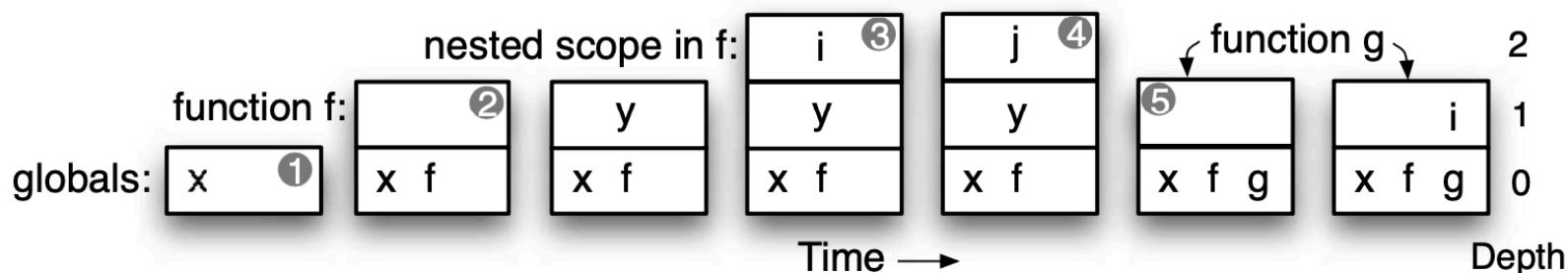
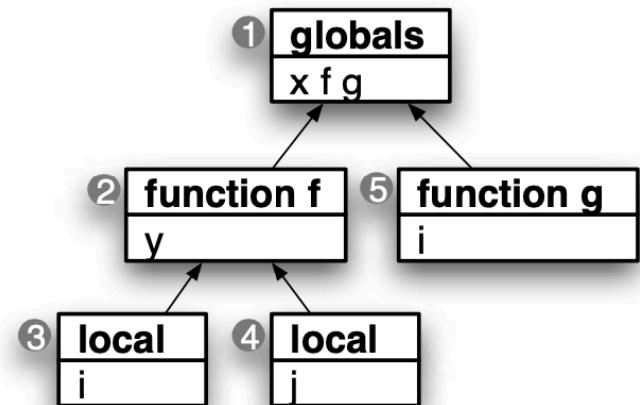
```
public interface Scope {  
    public String getScopeName();          // do I have a name?  
    public Scope getEnclosingScope();      // am I nested in another?  
    public void define(Symbol sym);       // define sym in this scope  
    public Symbol resolve(String name);   // look up name in scope  
}
```

# Definición y conceptos - Alcance

- Los lenguajes de programación también permiten anidar un alcance dentro de otro.
- Para rastrear alcances anidados, metemos y sacamos alcances de una pila.
- Al encontrar un nuevo alcance, lo metemos a la pila y al salir de La cima de la pila se llama alcance actual.
- La cima de la pila se llama alcance actual.

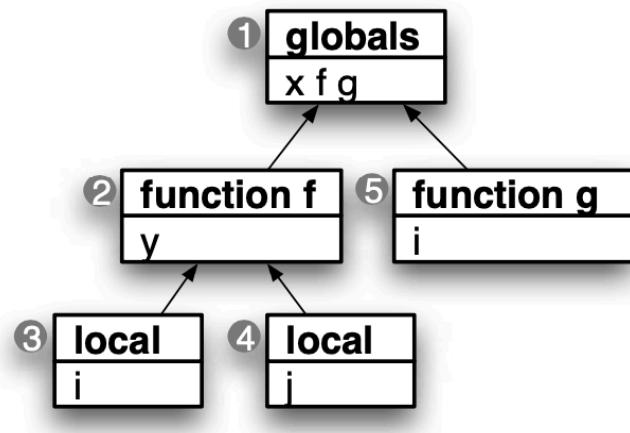
# Definición y conceptos - Alcance

```
// start of global scope
int x;          // define variable x in global scope
void f() {      // define function f in global scope
    int y;      // define variable y in local scope of f
    { int i; } // define variable i in nested local scope
    { int j; } // define variable j in another nested local scope
}
void g() {      // define function g in global scope
    int i;      // define variable i in local scope of g
}
```



# Definición y conceptos - Alcance

- Para mantener todos los símbolos en su alcance correspondiente, utilizamos una estructura de datos llamada árbol de alcance (scope tree) que funciona como una colección de pilas.
- Cada trayectoria desde un nodo hacia la raíz representa una pila de alcances.
- Por ejemplo, el nodo 4 tiene una pila de alcances implícita 4 2 1.
- Los niveles del árbol corresponden a los niveles de anudamiento de los alcances.



# Definición y conceptos - Alcance

- **Alcance monolítico.** Los primeros lenguajes de programación como BASIC tiene un alcance global. Hoy en día, sólo los lenguajes simples tienen alcance sencillo.

```
host=antlr.org          # define properties in single, global scope
port=80                # a set can act as a symbol table
webmaster=parrt@antlr.org
```

- El rastreo de símbolos para un alcance monolítico significa mantener un conjunto de símbolos.
- Conforme vamos encontrando definiciones, creamos símbolos y los agregamos al conjunto. Después podemos acceder a los símbolos a través de su nombre.
- **Alcance múltiple y anidamiento.** El alcance múltiple nos permite reutilizar el mismo nombre en diferentes regiones de código para identificar diferentes entidades de programa.

```
void f() { int x; printf(x); } // this x lives in f's scope
void g() { float x; printf(x); } // this x lives in g's scope
```

# Definición y conceptos - Alcance

- Para construir un árbol de alcance se requiere ejecutar una secuencia de las operaciones: push, pop y def.
  - Push. Al inicio de un alcance, meter (push) un nuevo alcance a la pila. Esta operación es equivalente a “agregar hijo”.

```
// create new scope whose enclosing scope is the current scope
currentScope = new LocalScope(currentScope);      // push new scope
```

- Pop. Al final de un alcance, sacar (pop) el alcance actual de la pila, tomando el alcance anterior como el actual.

```
currentScope = currentScope.getEnclosingScope(); // pop scope
```

- Def. Definir un símbolo en el alcance actual.

```
Symbol s = «some-new-symbol»;
currentScope.define(s); // define s in current scope
```

# Definición y conceptos - Alcance

- Tomemos como ejemplo el árbol de alcance visto anteriormente, las operaciones que tendría que realizar un pares para construir dicho árbol son:
  1. push global scope ①.
  2. def variable x in current scope, ①.
  3. def method f in scope ① and push scope ②.
  4. def variable y.
  5. push local scope ③.
  6. def variable i.
  7. pop ③ revealing ②.
  8. push local scope ④.
  9. def variable j.
  10. pop ④ revealing ②.
  11. pop function f scope ② revealing ①.
  12. def method g in scope ① and push scope ⑤.
  13. def variable i.
  14. pop function g scope ⑤ revealing ①.
  15. pop global scope ①.

# Definición y conceptos - Alcance

- Cuando vemos una referencia a un símbolo x en un programa, nuestro cerebro inconscientemente busca la definición más cercana. En otras palabras, nuestro cerebro trata de resolver (identificar) a que entidad del programa se refiere.
- Es fácil resolver un símbolo cuando se tiene sólo un alcance.
- El método resolve() lo único que hace es buscar el símbolo en el diccionario de alcance.

```
myOnlyScope.resolve(<<symbol-name>>);
```

# Definición y conceptos - Alcance

- Cuando existe más de un alcance, resolver un símbolo depende de la localización de la referencia al símbolo, ya que el mismo símbolo puede referirse a dos entidades de programa diferentes dependiendo del lugar en el código fuente donde aparece.
- Una pila de alcances de referencias es el conjunto de alcances en la trayectoria hacia la raíz del árbol de alcances. A esta pila se le llama el contexto semántico.
- Para resolver un símbolo, lo buscamos en su contexto semántico, iniciando en el alcance actual.

```
public Symbol resolve(String name) {  
    Symbol s = members.get(name);      // look in this scope  
    if ( s!=null ) return s;           // return it if in this scope  
    if ( enclosingScope != null ) { // have an enclosing scope?  
        return enclosingScope.resolve(name); // check enclosing scope  
    }  
    return null; // not found in this scope or there's no scope above  
}
```

- No importa que tan complicado sea el árbol de alcance, siempre podemos resolver un símbolo utilizando el mismo código.

`currentScope.resolve(«symbol-name»);`

# Definición y conceptos - Alcance

- El apuntador **enclosingScope** le dice al método **resolve()** exactamente donde buscar a continuación. En vez de hacer un algoritmo que implemente las reglas para buscar, construimos una estructura de datos que lo integra.
- Veamos como el método resolve() maneja dos funcionalidades importantes de los alcances anidados: se pueden ver símbolos fuera del alcance y se pueden redefinir símbolos encontrados en alcances de afuera. En el ejemplo se visualiza como resolver en C++:

```
// start of global scope
int x;                      // define variable x in global scope
int y;                      // define variable y in global space
void f() {                  // define function f in global scope
    float x;                // redefine x as local variable, hiding outer x
    printf("%f", x); // x resolves to f's local
    printf("%d", y); // y resolves to global
    {int z;}          // local scope nested within f's local scope
    printf("%d", z); // z is no longer visible; static analysis ERROR!
}
```

# Patrones - Tabla de símbolos para alcance monolítico

## **Propósito.**

- Este patrón construye una tabla de símbolos para un lenguaje con sólo un alcance.
- Este patrón es útil para lenguajes de programación simples (sin funciones), archivos de configuración, pequeños lenguajes gráficos, etc.

# Patrones - Tabla de símbolos para alcance monolítico

## Discusión.

- La meta principal cuando se construye una tabla de símbolos es la construcción de un árbol de alcance.
- Para este caso el árbol de alcance es muy sencillo dado que es un sólo nodo (alcance global).
- La siguiente tabla indica como construir el alcance sencillo.

<b>Upon</b>	<b>Action(s)</b>
Start of file	push a GlobalScope. def BuiltInType objects for any built-in types such as int and float.
Declaration x	ref x's type (if any). def x in the current scope.
Reference x	ref x starting in the current scope.
End of file	pop the GlobalScope.

# Patrones - Tabla de símbolos para alcance monolítico

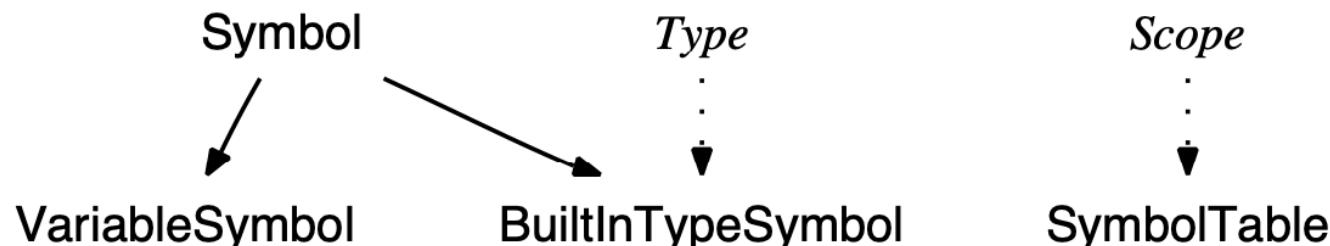
- Aplicando estos conceptos a las siguientes declaraciones en el lenguaje Cymbol, un subconjunto de C++.

```
int i = 9;  
float j;  
int k = i+2;
```

push global scope, def int, def float,

ref int, def i, ref float, def j, ref int, def k, ref i, pop global scope.

- Para poblar la tabla de símbolos, necesitamos objetos que representen las variables y tipos de datos predefinidos.
- Etiquetaremos los símbolos de tipo con la interfaz Type y haremos que la tabla de símbolos juegue el papel de alcance a través de la interfaz Scope.



# Patrones - Tabla de símbolos para alcance monolítico

## Implementación.

- Para el programa anterior en Symbol, debemos generar una salida que nos convenza que estamos manejando correctamente los símbolos.

```
$ java Test < t.symbol
line 1: ref int
line 1: def i
line 2: ref float
line 2: def j
line 3: ref int
line 3: ref to <i:int> // <i:int> means i has type int
line 3: def k
globals: {int=int, j=<j:float>, k=<k:int>, float=float, i=<i:int>}
$
```

# Patrones - Tabla de símbolos para alcance monolítico

- Primero definiremos los símbolos.

```
public class Symbol { // A generic programming language symbol
    String name;          // All symbols at least have a name
    Type type;
    public Symbol(String name) { this.name = name; }
    public Symbol(String name, Type type) {this(name); this.type = type;}
    public String getName() { return name; }
    public String toString() {
        if ( type!=null ) return '<'+getName()+":"+type+ '>';
        return getName();
    }
}
```

# Patrones - Tabla de símbolos para alcance monolítico

```
/** Represents a variable definition (name,type) in symbol table */
public class VariableSymbol extends Symbol {
    public VariableSymbol(String name, Type type) { super(name, type); }
}

/** A symbol for built in types such int, float primitive types */
public class BuiltInTypeSymbol extends Symbol implements Type {
    public BuiltInTypeSymbol(String name) { super(name); }
}

public interface Type { public String getName(); }
```

# Patrones - Tabla de símbolos para alcance monolítico

- Es necesario incluir la interface **Scope** que sirve para definir un alcance.

```
public interface Scope {  
    public String getScopeName();          // do I have a name?  
    public Scope getEnclosingScope();      // am I nested in another?  
    public void define(Symbol sym);        // define sym in this scope  
    public Symbol resolve(String name);    // look up name in scope  
}
```

# Patrones - Tabla de símbolos para alcance monolítico

- Ahora necesitamos un diccionario para mantener los símbolos (un árbol de sólo un nodo o una pila de alcance de profundidad 1). La llamaremos **SymbolTable**

```
import java.util.*;
public class SymbolTable implements Scope { // single-scope symtab
    Map<String, Symbol> symbols = new HashMap<String, Symbol>();
    public SymbolTable() { initTypeSystem(); }
    protected void initTypeSystem() {
        define(new BuiltInTypeSymbol("int"));
        define(new BuiltInTypeSymbol("float"));
    }
    // Satisfy Scope interface
    public String getScopeName() { return "global"; }
    public Scope getEnclosingScope() { return null; }
    public void define(Symbol sym) { symbols.put(sym.name, sym); }
    public Symbol resolve(String name) { return symbols.get(name); }

    public String toString() { return getScopeName()+"："+symbols; }
}
```

# Patrones - Tabla de símbolos para alcance monolítico

- Ejemplo del uso de tabla de símbolos en el parser de pseudocódigo.

```
public class PseudoParser {  
    ArrayList<PseudoLexer.Token> tokens;  
    int tokenIndex = 0;  
    SymbolTable tabla;  
  
    public boolean parse(ArrayList<PseudoLexer.Token> tokens) {  
        this.tokens = tokens;  
  
        tabla = new SymbolTable();  
        tabla.initTypeSystem();  
  
        System.out.println("\n***** Reglas de producción *****\n");  
        return programa();  
    }  
}
```

# Patrones - Tabla de símbolos para alcance monolítico

# Patrones - Tabla de símbolos para alcance monolítico

```
// variables <Tipo> : <Lista de variables>
private boolean declaracionVariables() {
    if (token("VARIABLES")) {
        // Guardar el tipo de dato (está en el token)
        if (tipo()) {
            // Verificar que el tipo de dato esté en la tabla de símbolos
            BuiltInTypeSymbol bit = tabla.resolve(tipo);
            if (bit == null)
                // Marcar error y terminar

            if (token("DOSPUNTOS")) {
                if (listaVariables()) {
                    // Guardar lista de variables
                    // Por cada variable crear un objeto VariableSymbol y guardarla en la tabla de símbolos
                    VariableSymbol vs = new VariableSymbol(nombre, bit);
                    tabla.define(vs);
                }
            }
        }
    }

    return false;
}
```

# Patrones - Tabla de símbolos para alcance monolítico

```
// <Asignacion> ::= <Variable> = <Expresion>
private boolean enunciadoAsignacion() {
    System.out.println("<Asignacion> --> <Variable> = <Expresion>");

    // Guardar nombre de la variable
    if (token("VARIABLE")) {
        // Verificar que la variable está declarada, si no marcar error
        VariableSymbol vs = tabla.resolve(nombre);
        if (vs == null)
            // Marcar error y terminar
        if (token("IGUAL")) {
            if (expresion()) {
                // Verificar que cada variable que aparezca en la expresión esté declarada
                return true;
            }
        }
    }

    return false;
}
```

# Patrones - Tabla de símbolos para alcance anidado

## **Propósito.**

- Este patrón rastrea símbolos y construye un árbol de alcance para lenguajes con alcance múltiple, posiblemente anidado..
- Las funciones en los lenguajes de programación son buen ejemplo de alcance anidado. Cada función tiene su propio alcance anidado en el alcance global o de la clase.

# Patrones - Tabla de símbolos para alcance anidado

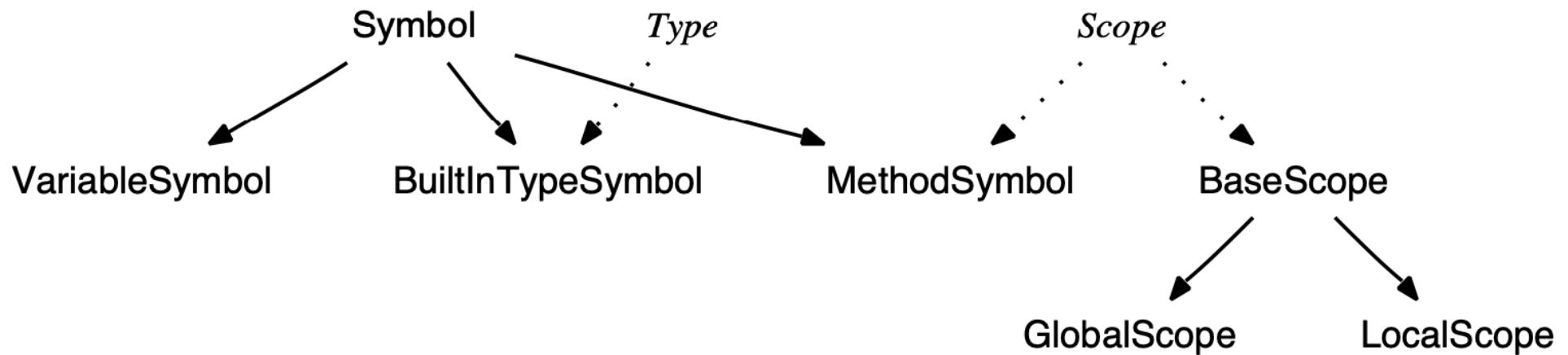
## Discusión.

- Para discutir sobre alcance anidado, utilizaremos funciones. Esto significa que necesitamos un símbolo función y alcances global, parámetros y local.

```
// start of global scope
int i = 9;
float f(int x, float y)
{
    float i;
    { float z = x+y; i = z; }
    { float z = i+1; i = z; }
    return i;
}
void g()
{
    f(i,2);
}
```

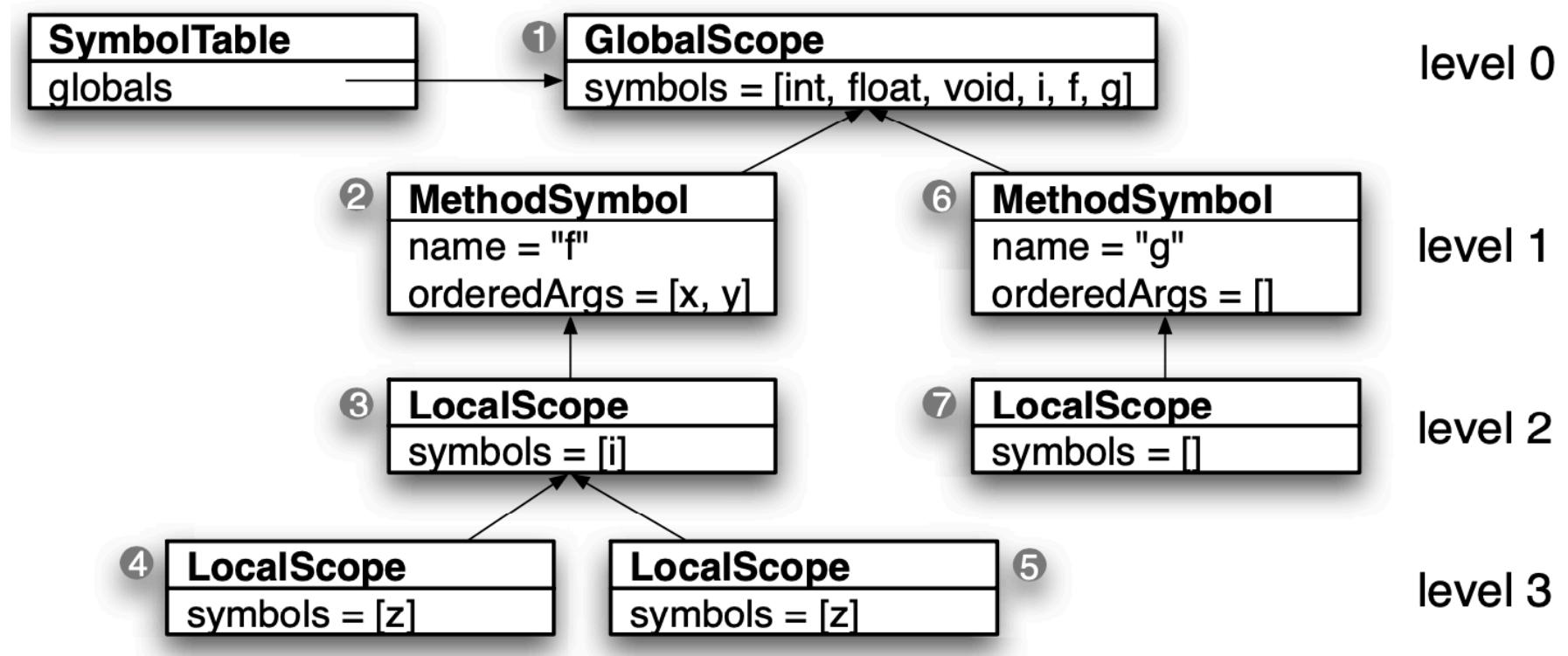
# Patrones - Tabla de símbolos para alcance anidado

- También necesitaremos más subclases de Symbol para manejar las funciones y alcance local.



# Patrones - Tabla de símbolos para alcance anidado

- Debemos notar que no hacemos distinción entre variables globales, parámetros y variables locales en términos de la lógica de la tabla de símbolos, todas son de objetos VariableSymbol. La única diferencia está en el alcance en el cual son definidas.



# Patrones - Tabla de símbolos para alcance anidado

- A continuación se muestran las reglas para construir un árbol de alcance a partir de alcances anidados y resolver símbolos en el contexto semántico correcto.

<b>Upon</b>	<b>Action(s)</b>
Start of file	push a GlobalScope. def BuiltInType objects for int, float, void.
Variable declaration x	ref x's type. def x in the current scope.
Method declaration f	ref f's return type. def f in the current scope and push it as the current scope.
{	push a LocalScope as the new current scope.
}	pop, revealing previous scope as current scope.
End of method	pop the MethodSymbol scope (the parameters).
Reference x	ref x starting in the current scope. If not found, look in the immediately enclosing scope (if any).
End of file	pop the GlobalScope.

# Patrones - Tabla de símbolos para alcance anidado

## **Implementación.**

- Para la implementación particular, es necesario llevar a cabo acciones para construir árboles de alcance mientras se recorre el árbol de sintaxis.
- Podemos disparar acciones directamente en el pares como se ha hecho en otros patrones.
- Iniciando con el código del patrón para tabla de símbolos para alcance monolítico, necesitamos llevar a cabo lo siguiente para agregar funciones y alcance anidado.

**1. Aumentar la sintaxis para apoyar el uso de funciones.**

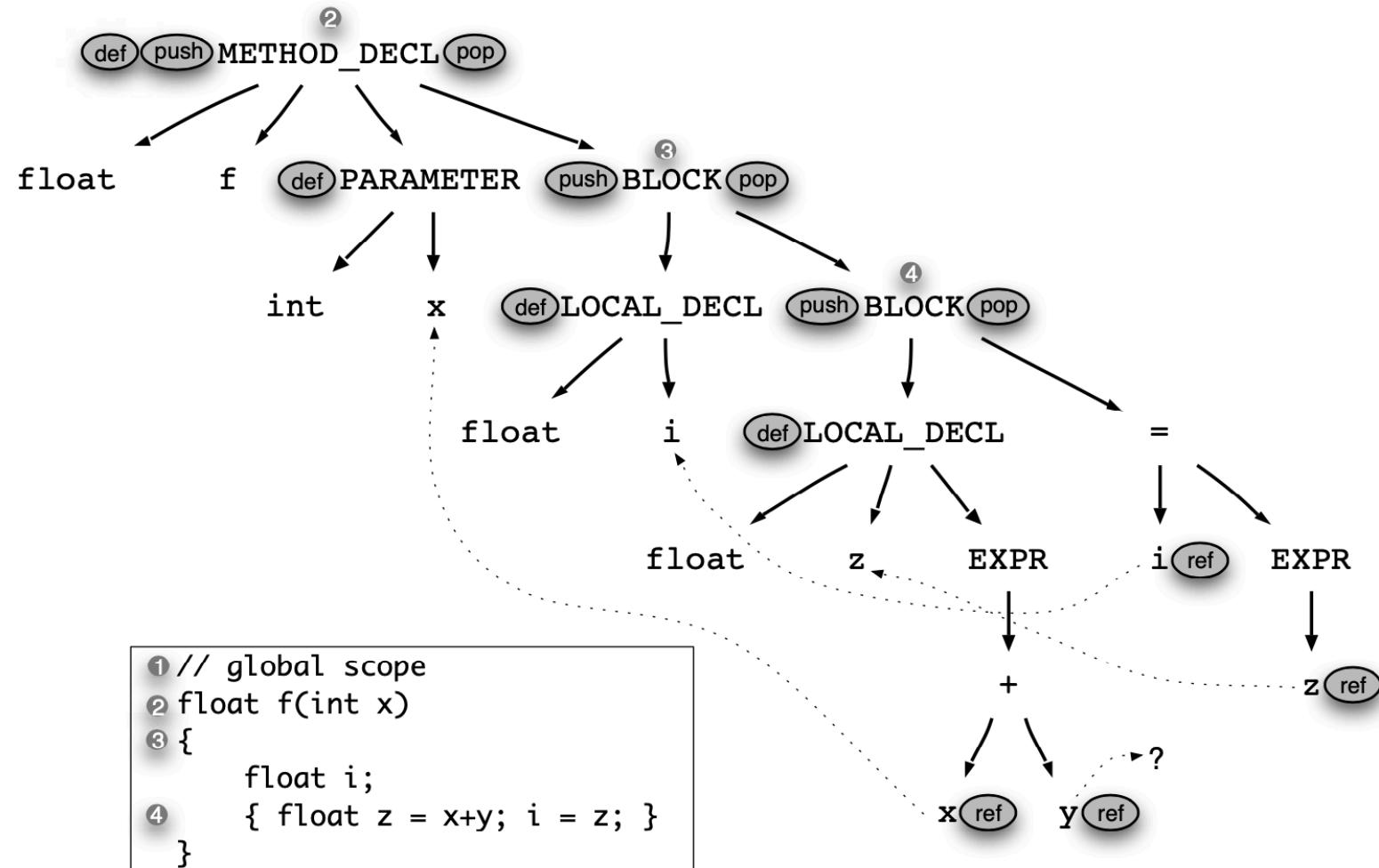
**2. Construir un árbol de sintaxis.**

**3. Definir un nuevo objeto tabla de símbolos.**

**4. Recorrer el árbol de sintaxis para poblar la tabla de símbolos y resolver referencias a variables y métodos.**

# Patrones - Tabla de símbolos para alcance anidado

- En el siguiente ejemplo, podemos ver donde y cuando necesitamos ejecutar las acciones **push**, **pop**, **def** y **ref**.



```
public abstract class BaseScope implements Scope {
    Scope enclosingScope;
    Map<String, Symbol> members = new HashMap<String, Symbol>();

    public BaseScope(Scope currentScope) {
        this.enclosingScope = currentScope;
    }

    public Scope getEnclosingScope() {
        return enclosingScope;
    }

    public void define(Symbol sym) {
        members.put(sym.name, sym);
    }

    public Symbol resolve(String name) {
        Symbol s = members.get(name);

        if (s != null)
            return s;

        if (enclosingScope != null)
            return enclosingScope.resolve(name);

        return null;
    }
}
```

```
public class LocalScope extends BaseScope {  
    public LocalScope(Scope currentScope) {  
        super(currentScope);  
    }
```

```
    public String getScopeName() {  
        return "Local";  
    }  
}
```

```
public class GlobalScope extends BaseScope {  
    public GlobalScope() {  
        super(null);  
    }
```

```
    public String getScopeName() {  
        return "Global";  
    }  
}
```

```
import java.util.HashMap;
import java.util.Map;

public class MethodSymbol extends Symbol implements Scope {
    Scope enclosingScope;
    Map<String, Symbol> members = new HashMap<String, Symbol>();

    public MethodSymbol(String name, VariableSymbol[] orderedArgs, Scope
        enclosingScope) {
        super(name);
        this.enclosingScope = enclosingScope;

        if (orderedArgs != null) {
            for (VariableSymbol v : orderedArgs) {
                define(v);
            }
        }
    }
}
```

```
public String getScopeName() {
    return name;
}

public Scope getEnclosingScope() {
    return enclosingScope;
}

public void define(Symbol sym) {
    members.put(sym.name, sym);
}

public Symbol resolve(String name) {
    Symbol s = members.get(name);

    if (s != null)
        return s;

    if (enclosingScope != null)
        return enclosingScope.resolve(name);

    return null;
}
```

```
public class AlcanceAnidado {
    public static void main(String args[]) {
        Scope currentScope;
        currentScope = new GlobalScope();
        currentScope.define(new BuiltInTypeSymbol("int"));
        currentScope.define(new BuiltInTypeSymbol("float"));
        currentScope.define(new BuiltInTypeSymbol("void"));
        // Colocar aqui el código restante de acuerdo al programa que se simulará
    }
}

① // start of global scope
② int i = 9;
③ float f(int x, float y)
④ {
⑤     float i;
⑥     { float z = x+y; i = z; }
⑦     { float z = i+1; i = z; }
⑧         return i;
⑨     }
⑩ void g()
⑪ {
⑫     f(i,2);
⑬ }

BuiltInTypeSymbol t1 = (BuiltInTypeSymbol) currentScope.resolve("int"); // int i
if (t1 == null) { } // Arrojar excepcion de tipo

currentScope.define(new VariableSymbol("i", t1));

BuiltInTypeSymbol returnType = (BuiltInTypeSymbol) currentScope.resolve("float");

if (returnType == null) { } // Arrojar excepcion de tipo

BuiltInTypeSymbol t2 = (BuiltInTypeSymbol) currentScope.resolve("int"); // int x
if (t2 == null) { } // Arrojar excepcion de tipo

BuiltInTypeSymbol t3 = (BuiltInTypeSymbol) currentScope.resolve("int"); // int y
if (t3 == null) { } // Arrojar excepcion de tipo

VariableSymbol parameters[] = { new VariableSymbol("x", t2), new VariableSymbol("y", t3)};
```

```
currentScope = new MethodSymbol("f", parameters, currentScope); // f(int x, int y)

t1 = (BuiltInTypeSymbol) currentScope.resolve("float"); // float i
if (t1 == null) { } // Arrojar excepcion de tipo

currentScope.define(new VariableSymbol("i", t1));

currentScope = new LocalScope(currentScope);

t1 = (BuiltInTypeSymbol) currentScope.resolve("float"); // float z = x + y
if (t1 == null) { } // Arrojar excepcion de tipo

currentScope.define(new VariableSymbol("z", t1));

VariableSymbol v1 = (VariableSymbol) currentScope.resolve("x");
if (v1 == null) { } // Arrojar excepcion de variable

VariableSymbol v2 = (VariableSymbol) currentScope.resolve("y");
if (v2 == null) { } // Arrojar excepcion de variable

v1 = (VariableSymbol) currentScope.resolve("i"); // i = z
if (v1 == null) { } // Arrojar excepcion de variable

v2 = (VariableSymbol) currentScope.resolve("z");
if (v2 == null) { } // Arrojar excepcion de variable

currentScope = currentScope.getEnclosingScope(); // Sale de alcance local
```