Declaración de símbolos y tipos de datos

3.2 Análisis semántico

En este apartado se describirá el trabajo que ha de desarrollar el estudiante para realizar el análisis semántico. Esta fase se encargará principalmente de:

- **Comprobación de la unicidad de declaraciones y definiciones**. No permitir que existan dos declaraciones con el mismo nombre en el mismo ámbito. (3.1.1 Análisis semántico, Página 25, Asimismo, debe comprobarse que no existen duplicaciones entre símbolos que pertenezcan al mismo ámbito o nombres de tipos. Cualquiera de estas violaciones o error de tipos constituye un error semántico que debe comunicarse al usuario.)

- **Comprobación de tipos**. Augurarse que todas las variables y constantes simbólicas referenciadas en el contexto de una expresión han sido previamente declaradas y que el tipo de cada construcción coincide con el previsto en su contexto.

- **Comprobación de concordancia en las referencias**. Asegurarse de que el tipo de las expresiones que sirven para referenciar el rango de un conjunto de tipo entero y que el rango cae dentro del rango de valores esperado según su declaración (cuando esto sea posible). En los registros comprobar que el acceso a un campo de un registro coincide con alguno de aquellos indicados en la declaración.

- **Comprobación de paso de parámetros**. Asegurarse de que el orden y tipo de los parámetros actuales pasados en una invocación a un subprograma coincide con el orden y tipo de los parámetros formales indicados en la declaración. En caso de funciones comprobar asimismo que el tipo de retorno coincide con el tipo esperado según la declaración.

- **Comprobación de la existencia de la sentencia de retorno en las funciones**. Asegurar que cada función contiene una sentencia de retorno.

Todas estas comprobaciones han de realizarse mediante acciones semánticas dentro de las reglas del analizador sintáctico. Para ello han de utilizarse las siguientes clases proporcionadas por el marco de trabajo tecnológico.

Declarar los símbolos correspondientes a cada ámbito en su correspondiente tabla de símbolos y los tipos de datos declarados en la tabla de tipos.

```
//Al crear un ámbito se crea automáticamente su tabla de símbolos y tipos
scopeManager.openScope(nombre)


//recuperamos el ámbito donde nos encontramos y obtenemos la tabla de símbolos y tipos
ScopeIF scope = scopeManager.getCurrentScope();
SymbolTableIF tablaSimbolos =scope.getSymbolTable();
TypeTableIF tablaTipos = scope.getTypeTable();


//Introducción en la Tabla de Símbolos el símbolo creado
SymbolConstant simboloConstante = new SymbolConstant(scope, nombre, tipo, valor.getLexema());
tablaSimbolos.addSymbol(simboloConstante);


//Introducción en la Tabla de Tipos el tipo creado
TypeRecord tipoRegistro = new TypeRecord(scope, id.getLexema());
tablaTipos.addType(id.getLexema(),tipoRegistro);



// No es lo mismo buscar en la pila de ámbito abierto que en la tabla de símbolos
// o en la tabla de tipos de un ámbito.
if (tablaSimbolos.containsSymbol(id.getLexema())) { //solo en la tabla de símbolos
if (scopeManager.containsSymbol(id.getLexema())) { //en todos los ámbitos que estén abiertos


if (tablaTipos.containsType(id.getLexema())) { //solo en la tabla de tipos
if (scopeManager.containsTypel(id.getLexema())) { //en todos los ámbitos que estén abiertos
```

Constantes simbólicas

NOMBRE_CONSTANTE = valor*;*

```
/* constantes simbólicas */
constantes
        MESES = 12;
        ABIERTO = cierto;
```

- Deberemos guardar en la tabla de símbolos del ámbito en el que se encuentra declarada el símbolo constante indicando: el ámbito, el nombre, el tipo y el valor.

- Para guardar el **valor** de la constante deberemos incluir en la clase **SymbolConstant.java** una variable con sus métodos de acceso o definir un nuevo constructor que incluya el valor de la constante.

C:\..\ArquitecturaPLII2020-2021\src\compiler\semantic\symbol\SymbolConstant.java

- Se emitirá un mensaje de error en el caso que se quiera guardar una constante cuyo identificador sea igual al de otro símbolo previamente guardado en la misma tabla de símbolos.

Ejemplo de una constante en el lenguaje cUNED (donde todas las constantes son del tipo primitivo ENTERO)

```
constante ::=  #DEFINE IDENTIFICADOR:id NUMEROENTERO:valor SEMICOLON
{:
//recuperamos el ámbito donde nos encontramos y obtenemos la tabla de símbolos
ScopeIF scope = scopeManager.getCurrentScope();
SymbolTableIF tablaSimbolos =scope.getSymbolTable();

//comprobamos que la constante no este ya contenida en la tabla de símbolos
if (tablaSimbolos.containsSymbol(id.getLexema())) {
        semanticErrorManager.semanticFatalError("Constante " + id.getLexema() + " en linea " + id.getLine() + " ya
        declarada");
} else {
        System.out.println("Constante " + id.getLexema() + " en linea " + id.getLine() + " aun No declarada");

        //introducimos en la tabla de símbolos la constante con su ámbito, nombre, tipo y valor
        TypeTableIF tablaTipos = scope.getTypeTable();
        TypeIF tipo = scopeManager.searchType("ENTERO");
        SymbolConstant simboloConstante = new SymbolConstant(scope, nombre, tipo, valor.getLexema());
        tablaSimbolos.addSymbol(simboloConstante);
}
:} ;
```

Declaración de tipos

- Al crear el ámbito programa principal, es decir el ámbito de nivel 0, se deben de crear los tipos de datos primitivos del lenguaje, que normalmente serán tipos de datos simples. El lenguaje PL1UnedES tiene dos tipos primitivos
    - Tipo entero
    - Tipo lógico (cierto y falso)
- Además de los tipos primitivos del lenguaje en la tabla de tipos deben de aparecer los diferentes tipos de datos definidos por el usuario o tipos compuestos.
- En lo que respecta al sistema de tipos, la equivalencia de éstos es nominal, no estructural. Es decir, dos tipos serán equivalentes únicamente si comparten el mismo nombre. No es posible definir dos tipos con el mismo nombre aunque pertenezcan a distintos ámbitos, si desde un ámbito se puede alcanzar el otro ámbito. Esto debe controlarlo el Analizador Semántico. (2.2.3.2 Tipos Compuestos, página 11).
- Se emitirá un mensaje de error en el caso que se quiera declarar un tipo cuyo identificador sea igual al de otro tipo previamente guardado en la pila de ámbitos abiertos, si es posible que existan dos tipos iguales en distinto ámbito que estén en el mismo nivel, por ejemplo en subprogramas que no estén anidados.
- También se emitirá un mensaje de error si el identificador del nuevo tipo es igual identificador de un símbolo previamente declarado en la tabla de símbolos del ámbito donde se declara el nuevo tipo.

Tipo Vector

nombreTipo = vector [n1..n2] de TipoPrimitivo;

tipos

TipoVectorEnteros = vector [1..3] de entero;
TipoVectorBooleanos = vector [1..3] de booleano;

Se debe de crear un nuevo tipo de datos, por ejemplo del tipo array o vector indicando el identificador del nuevo tipo y el ámbito donde se ha creado.

TypeArray tipoMatriz = new TypeArray (ambito, nombre);

C:\..\ ArquitecturaPLII2020-2021\src\compiler\semantic\type\TypeArray.java

- Es recomendable guardar el rango del vector, es decir los valores correspondientes a n1 y n2 dado que son constantes numéricas (literales o simbólicas). Así como el tipo primitivo del vector. Bien añadiendo un constructor que incluya además del ámbito y el nombre, su rango y tipo o creando los atributos y sus métodos de acceso en la clase TypeArray.java
- El analizador semántico comprobará que n2 es mayor o igual que n1.
- En el caso de ser una constante simbólica deberá comprobarse que esta declarada en la tabla de tipos de los ámbitos abiertos.

Variables

- Deberemos guardar en la tabla de símbolos del ámbito donde estén declaradas, el símbolo variable indicando el nombre de la variable, el ámbito y su tipo.

variables
nombre11, nombre 12, ..., nombre1N : Tipo1;
nombre21, nombre 22, ..., nombre2N : Tipo2;
...
nombreM1, nombre M2, ..., nombreMN : TipoM;

variables
x, z : INTEGER;
abierto : BOOLEAN;
v1, v2: TipoVector;
v : TipoVectorEnteros;
b : TipoVectorBooleanos;

SymbolVariable simboloVariable = new SymbolVariable (ambito, nombre, tipo);

C:\..\ArquitecturaPLII2020-2021\src\compiler\semantic\symbol\ SymbolVariable.java

- El analizador semántico comprobará que dentro de un mismo ámbito una variable solo se declara una vez. (2.2.4 Declaración de Variables, página 12).

- Se emitirá un mensaje de error en el caso que se quiera guardar una variable cuyo identificador sea igual al de otro símbolo previamente guardado en la tabla de símbolos o tabla de tipos.

- Si es posible que existan dos o más variables que tenga el mismo identificador pero siempre que estén en distinto ámbito.

- Ojo, “y” es una palabra reservada del lenguaje “Y lógica”.

Declaración de subprogramas

subprogramas

Procedimientos

procedimiento nombre (param11, param12,..., param1N : Tipo1;
param21, param22,..., param2N : Tipo2;
...
paramM1, paramM2,..., paramMN : TipoM):

procedimiento sumar ():
procedimiento sumar (a, b : entero):
procedimiento sumar (var a : entero; var v : booleano):

TypeProcedure tipoProcedimiento = new TypeProcedure(ambito, nombre);

C:\..\ArquitecturaPLII2020-2021\src\compiler\semantic\type\TypeProcedure.java

SymbolProcedure simboloProcedimiento = new SymbolProcedure (ambito, nombre, tipo);

C:\..\ArquitecturaPLII2020-2021\src\compiler\semantic\symbol\ SymbolProcedure.java

Funciones

funcion nombre (param11, param12,..., param1N : Tipo1;
param21, param22,..., param2N : Tipo2;
...
paramM1, paramM2,..., paramMN : TipoM): Tipo:

funcion sumar ( ) :entero:
funcion sumar (a, b : entero) : entero:
funcion mayorQue (var a : entero; var v :booleano) : booleano:

TypeFunction tipoFuncion = new TypeFunction(ambito, nombre);

C:\..\ArquitecturaPLII2020-2021\src\compiler\semantic\type\TypeFunction.java

SymbolFunction simboloFuncion = new SymbolFunction(ambito, nombre, tipo);

C:\..\ArquitecturaPLII2020-2021 \src\compiler\semantic\symbol\ SymbolFuction.java

- En la tabla de tipos habrá que registrar el tipo función o procedimiento correspondiente.
- Deberemos guardar en la tabla de símbolos del ámbito donde se declare el procedimiento o función, el símbolo función o procedimiento, indicando: su ámbito, su nombre, su tipo y sus parámetros y el tipo de retorno en caso de que se trate de un símbolo función. La lista de parámetros y el tipo de retorno también puede guardarse al registrar el tipo de dato en la tabla de tipos.
- Se emitirá un mensaje de error en el caso que se quiera guardar una función o procedimiento cuyo identificador sea igual al de otro símbolo o tipo previamente guardado en la tabla de símbolos o tipos del mismo ámbito.

Parámetros

param11, param12,..., param1N : Tipo1;
param21, param22,..., param2N : Tipo2;
...
paramM1, paramM2,..., paramMN

SymbolParameter simboloParametro = new SymbolParameter (ambito, nombre, tipo);

C:\..\ArquitecturaPLII2020-2021\src\compiler\semantic\symbol\ SymbolParameter.java

- Deberemos guardar en la tabla de símbolos del procedimiento o de la función, el símbolo parámetro indicando el ámbito, el nombre del parámetro y su tipo.

- Se emitirá un mensaje de error en el caso que se quiera guardar un parámetro cuyo identificador sea igual al de otro símbolo previamente guardado en la tabla de símbolos del procedimiento o de la función.

Ejemplo de un programa en PL1UnedES

```
programa Ejemplo:

    constantes MAX = 5;

    tipos Mivector = vector [1..MAX] de entero;

    variables i : entero;
        x : booleano;
        v1 : Mivector;

    subprogramas

    funcion EsMenorQueDos (a : entero): booleano:

        variables dos : entero;
            result : entero;
            esMayor : booleano;

        subprogamas

            procedimiento Imprime (numero:entero):
            comienzo
                escribir("imprimiento");
                escribir(numero);
            fin;


        comienzo
            dos = 2;
            result = a+dos;
            si result < 2 entonces:
                Imprime(result);
                esMenor = cierto;
            sino:
                esMenor=falso;
            fin si;
            devolver esMenor;
        fin;

    comienzo
        v1[1] = 1;
        v1[2] = 2;
        v1[3] = 3;
        para i en 1..3:
            x = esMenorQueDos(v1[i]);
        fin para;
    fin.
```

```
[java] [SEMANTIC DEBUG] - ** TYPE TABLES **
[java] [SEMANTIC DEBUG] - Type - TypeArray [scope = EJEMPLO, name = MIVECTOR] - {n1=1, n2=5,
tipoPrimitivo=ENTERO}
[java] [SEMANTIC DEBUG] - Type - TypeSimple [scope = EJEMPLO, name = ENTERO] - {}
[java] [SEMANTIC DEBUG] - Type - TypeFunction [scope = EJEMPLO, name = ESMENORQUEDOS] -
{tipoRetorno=LOGICO, listaParametros=[Symbol - SymbolParameter [scope = EJEMPLO, name = A, type = ENTERO]
- {}]}
[java] [SEMANTIC DEBUG] - Type - TypeSimple [scope = EJEMPLO, name = LOGICO] - {}
[java] [SEMANTIC DEBUG] - Type - TypeProcedure [scope = ESMENORQUEDOS, name = IMPRIME] - {
listaParametros=[Symbol - SymbolParameter [scope = ESMAYORQUEDOS, name = NUMERO, type = ENTERO] - {}]}
[java] [SEMANTIC DEBUG] - ** SYMBOL TABLES **
[java] [SEMANTIC DEBUG] - SYMBOL TABLE [EJEMPLO]
[java] [SEMANTIC DEBUG] - Symbol - SymbolConstant [scope = EJEMPLO, name = MAX, type = ENTERO] -
{valor=5}
[java] [SEMANTIC DEBUG] - Symbol - SymbolFunction [scope = EJEMPLO, name = ESMAYORQUEDOS, type =
ESMAYORQUEDOS] - {tipoRetorno=LOGICO, listaParametros=[compiler.syntax.nonTerminal.Parametro@27c170f0]}
[java] [SEMANTIC DEBUG] - Symbol - SymbolVariable [scope = EJEMPLO, name = X, type = LOGICO] - {}
[java] [SEMANTIC DEBUG] - Symbol - SymbolVariable [scope = EJEMPLO, name = I, type = ENTERO] - {}
[java] [SEMANTIC DEBUG] - Symbol - SymbolVariable [scope = EJEMPLO, name = V1, type = MIVECTOR] - {}
[java] [SEMANTIC DEBUG] - SYMBOL TABLE [ESMENORQUEDOS]
[java] [SEMANTIC DEBUG] - Symbol - SymbolParameter [scope = ESMENORQUEDOS, name = A, type = ENTERO] - {}
[java] [SEMANTIC DEBUG] - Symbol - SymbolProcedure [scope = ESMENORQUEDOS, name = IMPRIME, type =
IMPRIME] - {listaParametros=[compiler.syntax.nonTerminal.Parametro@76ed5528]}
[java] [SEMANTIC DEBUG] - Symbol - SymbolVariable [scope = ESMENORQUEDOS, name = ESMAYOR, type = LOGICO]
- {}
[java] [SEMANTIC DEBUG] - Symbol - SymbolVariable [scope = ESMENORQUEDOS, name = DOS, type = ENTERO] - {}
[java] [SEMANTIC DEBUG] - Symbol - SymbolVariable [scope = ESMENORQUEDOS, name = RESULT, type = ENTERO] -
{}
[java] [SEMANTIC DEBUG] - SYMBOL TABLE [IMPRIME]
[java] [SEMANTIC DEBUG] - Symbol - SymbolParameter [scope = IMPRIME, name = NUMERO, type = ENTERO] - {}
BUILD SUCCESSFUL
Total time: 4 seconds
```

Detección de Errores

Los errores semánticos deben ser errores fatales que aborten la ejecución del compilador, ver sección 3.2.1 del documento “DirectricesImplementacionPracticaPL2_2020-2021”.

Error al detectar un símbolo ya declarado en la tabla de símbolos de un ámbito:

```
programa Ejemplo;

      constantes MAX = 5;
            MAX = cierto;
```

```
finalTest:
     [java] [SYNTAX INFO] - Input file > C:\Users\josep\Documents\ArquitecturaPLIIGrupoA-2019-
2020\doc\test\test.muned
     [java] [SYNTAX INFO] - Output file > C:\Users\josep\Documents\ArquitecturaPLIIGrupoA-2019-
2020\doc\test\test.ens
     [java] [SYNTAX INFO] - Starting parsing...
     [java] Nuevo ambito abierto de nombre: EJEMPLO de nivel: 0 y su id es: 0
     [java] Constante MAYOR en linea 3 aun No declarada
     [java] [SEMANTIC FATAL] - Constante MAYOR en linea 4 ya declarada
     [java] Java Result: -1
BUILD SUCCESSFUL
Total time: 3 seconds
```

Error al detectar un tipo de dato ya declarado en la tabla de tipos con el mismo identificador aunque pertenezcan a distinto ámbito:

```
programa Ejemplo;

      constantes MAX = 5;

      tipos Mivector = vector [1..MAX] de entero;

      variables i : entero;
            x : booleano;
            v1 : Mivector;

      subprogramas

      funcion EsMenorQueDos (a : entero): booleano:

            variables dos : entero;
                  result : entero;
                  esMayor : booleano;

            subprogamas

                  procedimiento Imprime (numero:entero):
                  tipos Mivector = vector [1..100] de booleano;
                  comienzo
                        ...
                  fin;


            comienzo
                  ...
            fin;

      comienzo
            ...
      fin.
```

```
finalTest:
     [java] [SYNTAX INFO] - Input file > C:\Users\josep\Documents\ArquitecturaPLIIGrupoA-2019-
2020\doc\test\test.muned
     [java] [SYNTAX INFO] - Output file > C:\Users\josep\Documents\ArquitecturaPLIIGrupoA-2019-
2020\doc\test\test.ens
     [java] [SYNTAX INFO] - Starting parsing...
     [java] Nuevo ambito abierto de nombre: test de nivel: 0 y su id es: 0
     [java] Constante MAYOR en linea 3 aun No declarada
     [java] registro Campo del Registro casado tipo LOGICO desplazamiento 0
     [java] registro Campo del Registro edad tipo ENTERO desplazamiento 1
     [java] Tipo Vector MIVECTOR en linea 5 aun No declarado
     [java] ……
     [java] Nuevo ambito abierto de nombre: IMPRIME de nivel: 2 y su id es: 2
     [java] [SEMANTIC FATAL] - Tipo Vector MIVECTOR en linea 21 ya declarado
     [java] Java Result: -1
BUILD SUCCESSFUL
Total time: 3 seconds
```