Comprobación de tipos

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.

- **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.

Sentencia de asignación

referencia = expresion;

sentencia_Asignacion ::= referencia:ref ASIGNACION expresion:exp PUNTOYCOMA

- En las sentencias y expresiones es necesario no solo comprobar que los identificadores de las variables, o funciones están declarados en la tabla de símbolos sino que también existe compatibilidad de tipos.
- Comprobar que a la referencia se le puede asignar el valor de la expresión, es decir, se trata de una variable, parámetro, campo de registro o elemento de un vector.
- Comprobar el tipo de la referencia y de la expresión son compatibles.

```
programa ejemplo:

    constantes
        MAX = 18;
        FALSE = falso;
        TRUE = cierto;

    tipos
        Mivector = vector [1..5] de entero;

    variables
        abierto, cerrado : booleano;
        x, z: entero;
        v1, v2 : Mivector;

comienzo
  # las siguientes sentencias de asignación son compatibles en cuanto a tipos.
  x = 16;
  x = MAX;
  x = z;
  x = v1[3];
  abierto = TRUE;
  cerrado = abierto;
  x = v2[x];
  v1[2] = x;
  z = x * 5;
  z = (x+5);
  abierto = 5 == x;
  # las siguientes sentencias de asignación no son compatibles en cuanto a tipos
  # y debe de emitirse un error semántico.
  z = cierto;
  z = abierto;
  x = v1;
  abierto = v1[3];
  v1 = x;
  v1 = v2;

fin.
```

Expresiones

- Expresiones aritméticas deben de devolver un valor de tipo primitivo ENTERO
- Expresiones lógicas deben de devolver un valor de tipo primitivo LOGICO.

Expresiones aritméticas:

- Constantes literales de tipo entero
- Constantes simbólicas de tipo entero
- Los identificadores de variables o parámetros de tipo entero
- Las funciones que devuelve un valor de tipo entero
- Las expresiones aritméticas de la suma y la multiplicación de dos expresiones aritméticas.

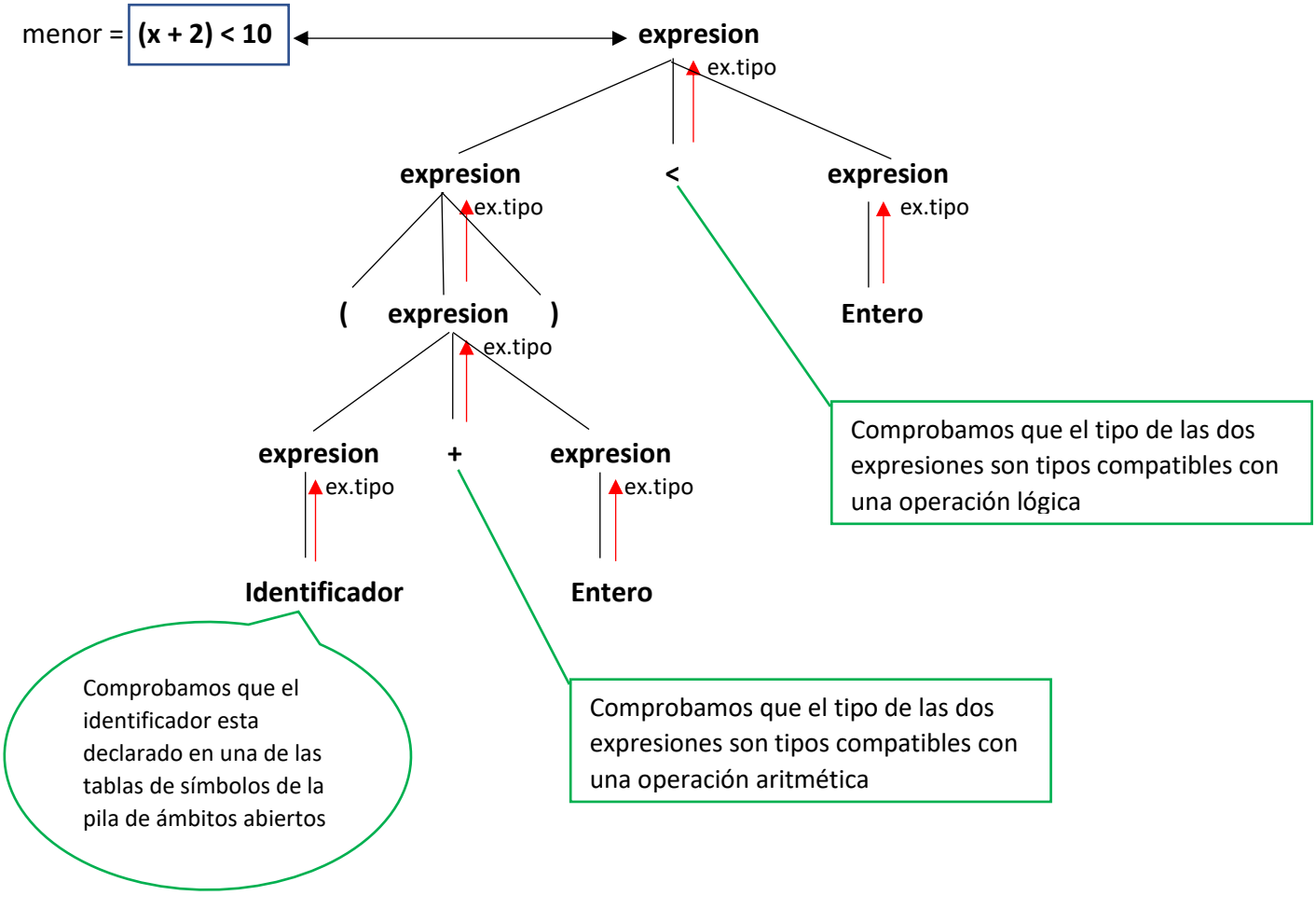
```
comienzo
   x = 16;
   x = MAX;
   x = z;
   x = v1[3];
   z = sumar (x, z);
   z = x + 5;
   z = (x + 5) * (x + 7);
fin.
```

Expresiones lógicas:

- Constantes literales de tipo lógico
- Constantes simbólicas de tipo lógico
- Los identificadores de variables o parámetros de tipo lógico
- Las funciones que devuelve un valor de tipo lógico
- Las expresión lógica de la negación (no) de un expresión lógica
- La conjunción lógica (Y) de dos expresiones lógicas.
- La expresión con operadores relacionales (<, ==) de dos expresiones aritméticas debe de dar como resultado una expresión lógica

```
comienzo
   abierto = TRUE;
   cerrado = abierto;
   abierto = v2[5];
   v = menorQue (x, z);
   v = no TRUE;
   v = abierto Y cerrado;
   cerrado = z < 5;
fin.
```

```
sentencia_Asignacion ::= IDENTIFICADOR:ref ASIGNACION expresion:ex PUNTOYCOMA
{:
        //recuperamos el ámbito donde nos encontramos
        ScopeIF scope = scopeManager.getCurrentScope();
        //comprobamos que la IDENTIFICADOR existe en las tablas de símbolos de los ámbito abiertos
        // y que se trata de una variable o un parametro de un subprograma
        if ((scopeManager.containsSymbol(ref.getLexema())) &&
((scopeManager.searchSymbol(ref.getLexema()) instanceof SymbolVariable) ||
(scopeManager.searchSymbol(ref.getLexema()) instanceof SymbolParameter))) {
                System.out.println("sentencia_asignacion - Referencia  " + ref.getLexema() + " en linea " +
ref.getLine() + " esta declarada");
        } else {
                semanticErrorManager.semanticFatalError("sentencia_asignacion - Referencia  " +
ref.getLexema() + " en linea " + ref.getLine() + " no esta declarada");
        }
        //Comprobamos que el tipo de la referencia y el de la expresion coinciden
        if (scopeManager.searchSymbol(ref.getLexema()).getType().getName().equals(ex.getTipo())){
                System.out.println("sentencia_asignacion coinciden los tipo");
        } else{
                semanticErrorManager.semanticFatalError("sentencia_asignacion no coinciden los tipos en la
linea " + ref.getLine());
        }
:}
```



Sentencia de llamada a un procedimiento

sentenciaProcedimiento ::= IDENTIFICADOR OBRACKET parámetros CBRACKET PUNTOYCOMA

- Comprobamos que el identificador del procedimiento existe en la pila de ámbitos abiertos, y que se trata de un símbolo SymbolProcedure y que su tipo es TypeProcedure*.
- Comprobamos que el número de parámetros del procedimiento que se llama (parámetros actuales) es igual a los parámetros formales (en número, tipo y orden) de la declaración del procedimiento.
- Paso por valor: solo se pueden pasar parámetros de tipo primitivo del lenguaje, es decir de tipo enteros o de tipo lógico, no se pueden pasar estructuras enteras como un vector.
- Paso por referencia: los parámetros no podrán ser expresiones, solo pueden ser referencias a variables y elementos de un vector. Por lo que se podrá pasar una variable entera o lógica y ¡¡¡también una estructura completa!!!, es decir un vector completo.

Invocación de una función

invocarFuncion ::= IDENTIFICADOR OBRACKET parámetros CBRACKET PUNTOYCOMA

referencia := invocarFuncion;

- Comprobamos que el identificador de la función existe en la pila de ámbitos abiertos, y que se trata de un símbolo SymbolFunction y que su tipo es TypeFunction*.
- Comprobamos que el número de parámetros de la función que se invoca (parámetros actuales) es igual a los parámetros formales (en número, tipo y orden) de la declaración de la función.
- Las funciones se invocarán como una expresión que tendrá su tipo de retorno y que debe de coincidir con el tipo de la referencia.
- Paso por valor: solo se pueden pasar parámetros de tipo primitivo del lenguaje, es decir de tipo enteros o de tipo lógico, no se pueden pasar estructuras enteras como un vector.
- Paso por referencia: los parámetros no podrán ser expresiones, solo pueden ser referencias a variables y elementos de un vector. Por lo que se podrá pasar una variable entera o lógica y ¡¡¡también una estructura completa!!!, es decir un vector completo.

Sentencia devolver (return)

devolver *expresión;*

- La comprobación de la existencia de **devolver** debe de realizarse dentro del análisis semántico en los subprogramas que se comporten como una función.
- El tipo de expresión de la sentencia **devolver** debe de coincidir con el tipo de retorno especificado en la cabecera de la función.
- Dentro del bloque de sentencias habrá que propagar un atributo que nos permita comprobar que en el bloque de sentencias de una función exista la propia sentencia **devolver**.
- Dentro del bloque de sentencias pueden existir más de una sentencia **devolver**.
- Lo normal es comprobar que hay una sentencia **devolver** por cada punto de salida, pero en la práctica lo limitamos a comprobar que por lo menos hay un **devolver**.

```
comienzo
        si (a < b) entonces:
                escribir (“b es mayor que a”);
                devolver a;
        sino:
                escribir (“a es mayor que b”);
                devolver b;
        fin si;
fin;
```

```
programa ejemplo:

    variables
        a,b,c : entero;
        v,f : booleano;

    subprogramas
        procedimiento imprimir (a : entero):
        comienzo
            escribir(a);
        fin;

        funcion sumar (var a: entero; b: entero): entero:
        variables
            x : entero;
        comienzo
            x = a + b;
            devolver x;
        fin;

comienzo
    imprimir (12);
     c = sumar(a, b);
    imprimir(sumar(a, b));

    # las siguientes sentencias no son compatibles en cuanto a tipos
    # o número de parámetros o invocación.
    v = sumar(a, b);
    c = sumar(a, v);
    c = sumar(12, 12);
    c = sumar(a);
    imprimir (12, 5);
    sumar(a,b);
    v = imprimir(12);
fin.
```

Sentencia de control de flujo condicional si – entonces - sino

si expresionLogica **entonces:** sentencias **fin si**;
si expresiónLogica **entonces:** sentencias **sino:** sentencias **fin si**;

```
si (a < b) entonces:
        escribir ("a es menor que b");
        devolver cierto;
sino:
        escribir ("b es menor o igual que a");
        devolver falso;
fin si;
```

- Comprobaremos que se trata de una expresión lógica
- El símbolo no terminal correspondiente a una expresión lógica arrastrará un atributo que nos permita comprobar si dicha expresión es lógica o no.
- Si se quiere comprobar que en el bloque de sentencias de la parte **si - entonces** contiene una sentencia **devuelve**, para que sea válido debe de aparecer también en el bloque de sentencias de la parte **sino** la sentencia **devuelve**, o existir una sentencia **devuelve** fuera del bloque **si – entonces - sino**.
- Si no hay parte **sino** aunque dentro del bloque de sentencias del **si-entonces** contenga una sentencia **devuelve**, como cabe la posibilidad que no llegue a ejecutarse el bloque **si-entonces** debe de existir una sentencia **devuelve** fuera del bloque de sentencias **si-entonces**.

Sentencia de control de flujo iterativo para

para indice **en** expresionComienzo . . expresionFinal :
        sentencias;

fin para;

```
para i en 1..v1[3]:

        escribir(a[i]);

fin para;
```

- Comprobaremos que la variable índice esta declarada en la tabla de símbolos abiertos.
- Comprobamos que el tipo de la variable índice es de tipo primitivo entero
- Comprobamos que expresionComienzo y expresionFinal son expresiones aritméticas, es decir deben de ser de tipo entero.
- Aunque dentro del bloque de sentencias del **para** contenga una sentencia **devuelve**, como cabe la posibilidad de que el bucle para no llegue a ejecutarse debe de existir una sentencia **devuelve** fuera del bloque de sentencias del bucle **para**.

Expresiones de acceso a vectores

variable[indice] := expresion;

referencia := variable[índice];

- Debe de comprobarse que el índice es una expresión aritmética.
- Al poder usarse expresiones aritméticas para acceder a un elemento de un vector, la comprobación de que no supere al índice de la declaración se debería hacer en tiempo de ejecución. Para simplificar el desarrollo de la práctica, no se tiene que detectar este tipo de errores. En caso de acceder mediante constantes enteras literales, sí es posible realizar esa comprobación en la fase de análisis semántico, y de hecho se debe de hacer en la práctica.
- Comprobar que coincide el tipo del vector con el tipo de la expresión o el tipo de la referencia con el tipo del vector que hay que asignar.

```
programa ejemplo:

    constantes
        MAX = 18;
        FALSE = falso;
        TRUE = cierto;

    tipos
        Mivector = vector [1..5] de entero;
        Tuvector = vector [1..10] de booleano;

    variables
        abierto, cerrado : booleano;
        x, z: entero;
        v1, v2 : Mivector;
        b1, b2 : Tuvector;

 comienzo
    x = v1[3];
    cerrado = b1[9];
    x = v2[x];
     v1[2] = x;
    v1[3] = (x+v2[1]) * v1[v2[z]];
    cerrado = v1[3] < MAX;
    b1[5] = no FALSE;
    b2[6] = b1[2] Y (v1[3] < MAX);
    v1[3] = v1[v2[v2[z]]];
fin.
```

Sentencia de salida

escribir(parametro);

escribir(“cadena de caracteres”);

- No es necesario comprobar que parámetro es una cadena, ya viene definido por la propia regla sintáctica del lenguaje.

escribir(expresion);

- Comprobar que expresión sea del tipo entero, es decir que se trate de una expresión aritmética.

escribir();

- No es necesario comprobar, ya viene definido por la propia regla sintáctica del lenguaje.

```
comienzo
        escribir(“el resultado es: ”);
        escribir();
        escribir((a+5)*sumar(v1[3],z));
fin.
```