Índice

Marina de la Cruz
Alfonso Ortega

Índice

Índice

Notación

- En este documento, la representación de las producciones de la gramática del lenguaje de programación ÓMICRON se ajusta a la notación propia de Bison con las siguientes particularidades:
  - Los símbolos no terminales se representan en minúsculas
  - Los símbolos terminales se representan con TOK_ …
  - Los símbolos terminales de un carácter se representan encerrados entre comillas simples

Objetivo del análisis semántico

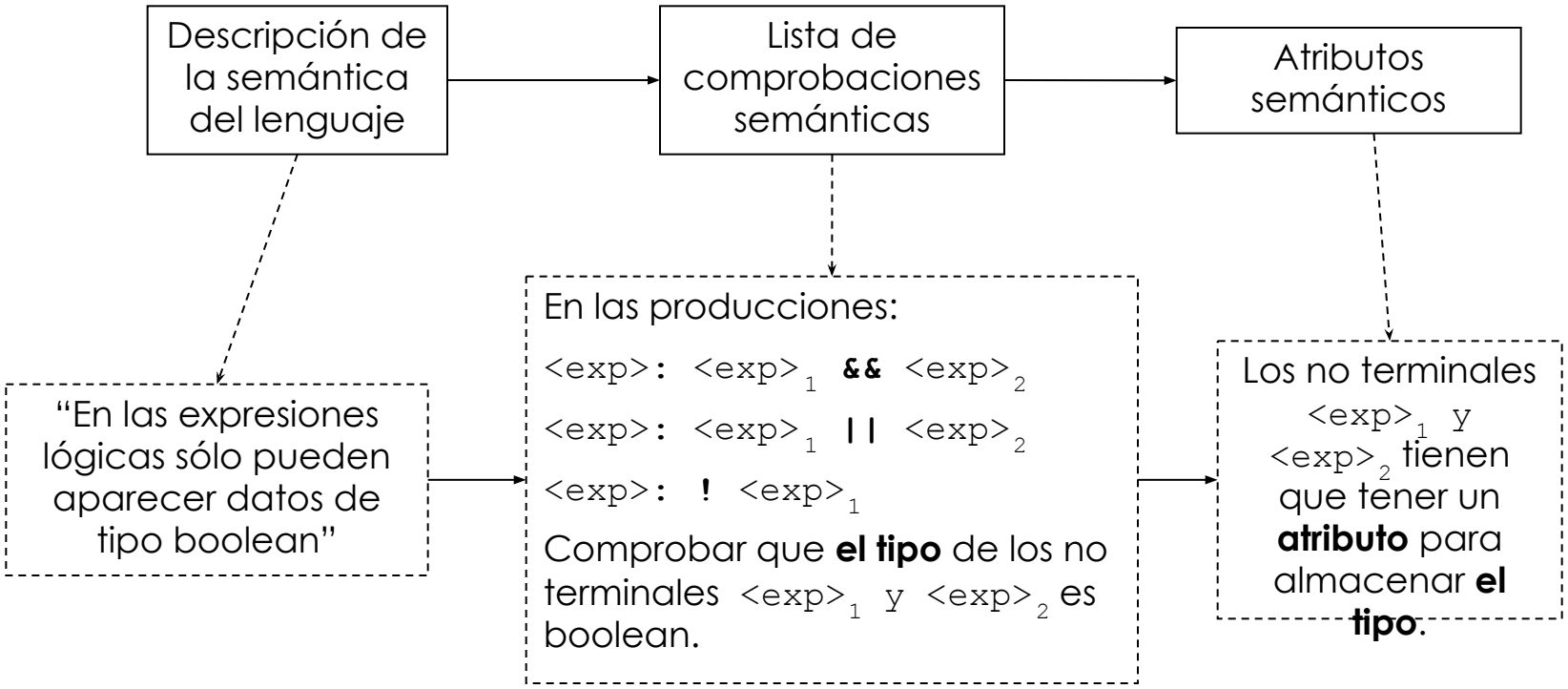
- La semántica del lenguaje forma parte de la especificación del mismo. Normalmente **la semántica de un lenguaje se describe de manera informal**.

- El objetivo del análisis semántico es **comprobar si la semántica del programa que se está compilando cumple las especificaciones de la semántica del lenguaje fuente**. Algunas de estas comprobaciones son:
  - comprobación de tipos en sentencias de asignación
  - comprobación de tipos en operaciones aritmético-lógicas
  - comprobación de tipos en las sentencias condicionales
  - comprobación de la declaración de las variables antes de su uso
  - comprobación del número de parámetros en la llamada a una función

- Es imprescindible conocer las restricciones semánticas del lenguaje antes de iniciar su desarrollo (leer el enunciado con detenimiento)

Pasos para la construcción de un analizador semántico

- Para realizar el análisis semántico se utilizan **gramáticas de atributos.** (Gramática independiente del contexto + sistema de atributos)
- Hay que **definir los atributos** (los necesarios para realizar las comprobaciones semánticas y también la generación de código)



Pasos para la construcción de un analizador semántico

- Además de definir los atributos hay que definir el modo en que se **calculan los valores de los atributos**.

- Por ejemplo, el atributo tipo del no terminal <exp> se calcula/evalúa en muchas producciones. Algunas de ellas son:

  - En la producción `<exp>: <exp>`$_1$ **&&** `<exp>`$_2$ al atributo **tipo** del no terminal <exp> se le asigna el valor boolean.
  - En la producción `<exp>: <exp>`$_1$ **||** `<exp>`$_2$ al atributo **tipo** del no terminal <exp> se le asigna el valor boolean.
  - En la producción `<exp>:` **!** `<exp>`$_1$ al atributo **tipo** del no terminal <exp> se le asigna el valor boolean.

- Los pasos para la construcción del analizador semántico con Bison son:
  - De la especificación del lenguaje se deduce la lista de comprobaciones semánticas.
  - Los **atributos necesarios** se deducen a partir de las comprobaciones semánticas. El tipo de los atributos se define mediante la directiva **%union** dentro del fichero de especificación de Bison.
  - Los atributos son **sintetizados**.
    Esta restricción viene impuesta porque el tipo de analizador sintáctico que genera Bison es ascendente LALR(1).
    Si la gramática de atributos que se diseña para implementar el análisis semántico requiere atributos heredados, hay que definir un mecanismo para incorporar el cálculo de dichos atributos en el analizador ascendente generado por Bison.
  - El **cálculo de los atributos** se realiza en las acciones asociadas a las reglas.
  - El **acceso a los atributos** se realiza a través de las pseudovariables $$, $1, $2, ...
- En resumen, el analizador semántico se **incorpora al analizador sintáctico** construido con Bison **insertando en las acciones de las reglas las comprobaciones propias del análisis semántico** (posteriormente también se incorporará la generación de código).

Definición de los atributos semánticos

- Para la implementación del compilador de ÓMICRON se *propone*[1] el siguiente conjunto de atributos:
  - **char* lexema**: guarda el lexema de los identificadores.
  - **int tipo:** guarda el tipo de una expresión (TIPO_ENTERO, TIPO_LOGICO) Definir "omicron.h".
  - **int valor_entero**: guarda el valor de una constante entera.
  - **int es_direccion**: atributo que indica si un símbolo representa una dirección de memoria o es un valor constante.
  - **int etiqueta**: atributo necesario para gestión de sentencias condicionales e iterativas. Es un atributo definido exclusivamente para la generación de código.

- IMPORTANTE: no confundir la información guardada en la tabla de símbolos con la información que se guarda en el sistema de atributos.

[1] La elección de los atributos no es única, existen distintas alternativas.

Definición de los atributos semánticos

- Los **atributos semánticos** de los **terminales** los proporciona (calcula) el analizador léxico de la siguiente manera:
  - los **identificadores** tienen como valor semántico su **lexema**.
  - las **constantes enteras** tienen como valor semántico su **valor numérico**.

- Los **atributos semánticos** de los **no terminales** se calculan en el proceso de análisis sintáctico/semántico. Cada vez que se reduce una producción, se pueden calcular los atributos semánticos del símbolo no terminal de la parte izquierda de la producción.

Definición de los atributos semánticos

- En Bison, los atributos semánticos se definen con la declaración **%union** en el fichero de especificación.

- La declaración %union de Bison se transforma en una definición de tipo union de C. Este tipo de datos de C sólo permite el uso de uno de sus campos, y no es válido para símbolos con múltiples atributos. Para permitir la **multiplicidad de atributos**, se puede definir en la declaración %union un sólo campo (atributos) cuyo tipo (tipo_atributos) sea un tipo definido como struct con tantos campos como sea necesario para contemplar todos los casos posibles de valores semánticos de cualquier símbolo del lenguaje (terminal o no terminal).

NOTA: El tipo "tipo_atributos" se puede definir en un fichero de cabecera aparte.

Definición de los atributos semánticos

- En la declaración **%union** del fichero **omicron.y**, se define un sólo campo, **atributos**, cuyo tipo sea **tipo_atributos** (definido en el fichero omicron.h).
- Se cualifican con **<atributos>** las declaraciones **%token** de los símbolos terminales que tienen valor semántico.
- Se añaden las correspondientes declaraciones **%type** para los **no terminales** que tengan **atributos semánticos.**

omicron.y

```
%{
#include "omicron.h"
...
%}
%union
{
  tipo_atributos atributos;
}

%token <atributos> TOK_IDENTIFICADOR
%token <atributos> TOK_CONSTANTE_ENTERA


/* resto de los tokens sin valor */
/* semántico                     */

%type <atributos> exp
%type <atributos> comparacion

/* resto de los no terminales */
/* con atributos semánticos   */

...

%%
...
%%
...
```

Definición de los atributos semánticos

- En el fichero **omicron.h** se define el tipo **tipo_atributos** como un struct con los siguientes campos :
  - **lexema**: para identificadores.
  - **tipo**: para comprobación de tipos básicos.
  - **valor_entero**: para constantes enteras.
  - **es_direccion**: para controlar lo que representa cada símbolo (una dirección de memoria o un valor constante).
  - **etiqueta**: para la generación de código de sentencias condicionales e iterativas.

omicron.h

```
#ifndef _OMICRON_H
#define _OMICRON_H

#define MAX_LONG_ID 100
#define MAX_TAMANIO_VECTOR 64
...

/* otros defines */


typedef struct
{
   char lexema[MAX_LONG_ID+1];
   int tipo;
   int valor_entero;
   int es_direccion;
   int etiqueta;
}tipo_atributos;

#endif
```

- El fichero **omicron.l** se modifica para que el analizador léxico actualice los atributos semánticos de los terminales.

Descripción general

- **Descripción de la semántica:** los identificadores tienen que ser declarados antes de su uso y deben ser únicos en su ámbito. Es decir:
  - Las **variables (globales y locales)** deben ser declaradas antes de su uso y además ser únicas en su ámbito de declaración.
  - Las **funciones** deben ser declaradas antes de su uso y ser únicas en su ámbito de declaración.
  - Los **parámetros** formales de las funciones son tratados como variables locales de la función y como tales deben ser únicos en su ámbito de declaración. Son declarados implícitamente antes de su uso ya que aparecen en la cabecera de la función.

- **Comprobaciones semánticas:** se necesitan dos acciones diferentes:
  - En la **declaración** de una variable/función/parámetro hay que comprobar que no existe dentro de su ámbito otra variable/función/parámetro con el mismo identificador antes de registrarla en el sistema, es decir, guardarla en la tabla de símbolos.
  - En el **uso**/aparición de una variable/función/parámetro hay que comprobar que previamente ha sido declarada, es decir, que está guardada en la tabla de símbolos.

Gestión de ámbitos

- La gestión de ámbitos anidados se puede implementar utilizando una pila de tablas de símbolos. Si el número máximo de ámbitos es 2, también se puede usar 2 tablas para gestionar los ámbitos.

- En la compilación, es necesario realizar las siguientes acciones:

  - **Antes de la primera declaración del programa fuente se debe inicializar la tabla de símbolos**. Este proceso de inicialización implica crear una tabla de símbolos vacía correspondiente al ámbito principal. La inicialización se puede realizar en uno de los dos puntos siguientes:
    - En el programa principal antes de invocar a la función de análisis sintáctico.
    - En la producción del axioma antes de la sección de declaraciones. En cualquiera de los puntos ·

      programa: · TOK_MAIN · '{' · declaraciones funciones sentencias '}'

      Se crea un nuevo no terminal "ficticio" y se añade una producción lambda en cuya acción semántica se inicializa la tabla de símbolos. Por ejemplo:

      ```
      programa: inicioTabla TOK_MAIN '{' declaraciones funciones sentencias '}'
      inicioTabla: {
            /* Acciones de inicialización de la tabla de símbolos */
          }
      ```

Gestión de ámbitos

- **En la declaración de una función hay que abrir un nuevo ámbito**, el correspondiente a la función. Se verá con detalle más adelante.

- **Al terminar la compilación**, liberar los recursos de la tabla de símbolos. Se puede hacer en:
  - El programa principal, cuando termina la función de análisis sintáctico.
  - En el fichero de especificación Bison. En este caso habría que contemplar las dos posibles finalizaciones de la función de análisis, con error o sin error. Si no hay error se puede hacer la liberación en la producción del axioma, al final de la misma. Si hay error se podría hacer en la función yyerror.

Inserción en la tabla de símbolos

- Las declaraciones de **variables globales** y **locales** se ajustan a la sintaxis definida por las siguientes producciones de la gramática de ÓMICRON:

  - programa : TOK_MAIN ‘{‘ declaraciones funciones sentencias ‘}’
  - declaraciones : declaracion
  - | declaracion declaraciones

  - declaracion : modificadores_acceso clase identificadores ‘;’

  - identificadores : identificador
  - identificadores : identificador **‘,’** identificadores

  - funcion : TOK_FUNCTION modificador_acceso tipo_retorno identificador '('
    parametros_funcion ')' '{' declaraciones_funcion sentencias '}'

  - declaraciones_funcion : declaraciones
  - |

  - identificador : TOK_IDENTIFICADOR

Inserción en la tabla de símbolos

- Los **parámetros** de las **funciones** se ajustan a la sintaxis definida por las siguientes producciones de la gramática de ÓMICRON:
  - funcion : TOK_FUNCTION tipo identificador '(' parametros_funcion ')'
    '{' declaraciones_funcion sentencias '}'
  - parametros_funcion : parametro_funcion resto_parametros_funcion
  - |
  - resto_parametros_funcion : ';' parametro_funcion resto_parametros_funcion
  - |
  - parametro_funcion : tipo identificador
  - identificador : TOK_IDENTIFICADOR

- La declaración de **funciones** tiene la siguiente sintaxis:
  - funcion : TOK_FUNCTION tipo identificador '(' parametros_funcion ')'
    '{' declaraciones_funcion sentencias '}'
  - identificador : TOK_IDENTIFICADOR

Inserción en la tabla de símbolos

- Del estudio de las producciones relacionadas con la declaración de variables globales y locales, parámetros y funciones, se puede deducir que **cada vez que se realiza una declaración se reduce la producción siguiente**:
  - identificador : TOK_IDENTIFICADOR

Por lo tanto, **parece que el punto adecuado para insertar los identificadores de las variables globales y locales, parámetros y funciones en la tabla de símbolos es la acción semántica de la producción anterior.** No obstante, veremos que es más adecuado utilizar esta producción únicamente para variables locales y globales, y usar otros mecanismos para la inserción de los identificadores de parámetros y funciones.

Inserción en la tabla de símbolos

- Del estudio de las producciones relacionadas con la declaración de variables globales y locales, parámetros y funciones, se puede deducir que **cada vez que se realiza una declaración se reduce la producción siguiente**:
  - identificador : TOK_IDENTIFICADOR

Por lo tanto, **parece que el punto adecuado para insertar los identificadores de las variables globales y locales, parámetros y funciones en la tabla de símbolos es la acción semántica de la producción anterior.** No obstante, veremos que es más adecuado utilizar esta producción únicamente para variables locales y globales, y usar otros mecanismos para la inserción de los identificadores de parámetros y funciones.

MODIFICAR SÓLO PARA VAR GLOBAL Y LOCAL

Inserción en la tabla de símbolos

- Independientemente del punto seleccionado para la inserción de un identificador en la tabla de símbolos, es necesario **disponer** en dicho punto de **toda la información asociada al identificador**, como por ejemplo:
  - **lexema**
  - **categoría** (variable, parámetro o función)
  - **clase** (escalar o vector)
  - **tipo** (entero o booleano)
  - **tamaño** (número de elementos de un vector)
  - **número de variables locales**
  - **posición de variable local**
  - **número de parámetros**
  - **posición de parámetro**

- **En el momento de la inserción en la tabla de símbolos se debe comprobar la unicidad** de los identificadores.

Inserción en la tabla de símbolos

- Los objetivos del análisis semántico en el procesamiento de la **declaración de identificadores** son los siguientes:
  - **Insertar en la tabla de símbolos cada identificador de variable global o local, parámetro o función con su correspondiente información asociada.**
  - **Controlar la unicidad de identificadores** (haciendo uso de la tabla de símbolos).
- Para la inserción de cualquier identificador es necesario definir:
  - El punto en el que se realiza la inserción.
  - Los mecanismos que permiten disponer en ese punto de la información asociada al identificador.
- Se distinguirán tres casos diferentes:
  - Inserción de identificadores de variables globales y locales.
  - Inserción de identificadores de parámetros de funciones.
  - Inserción de identificadores de funciones.

Introducción

- El punto adecuado para insertar los identificadores de variables globales y locales en la tabla de símbolos es la acción semántica de la producción:

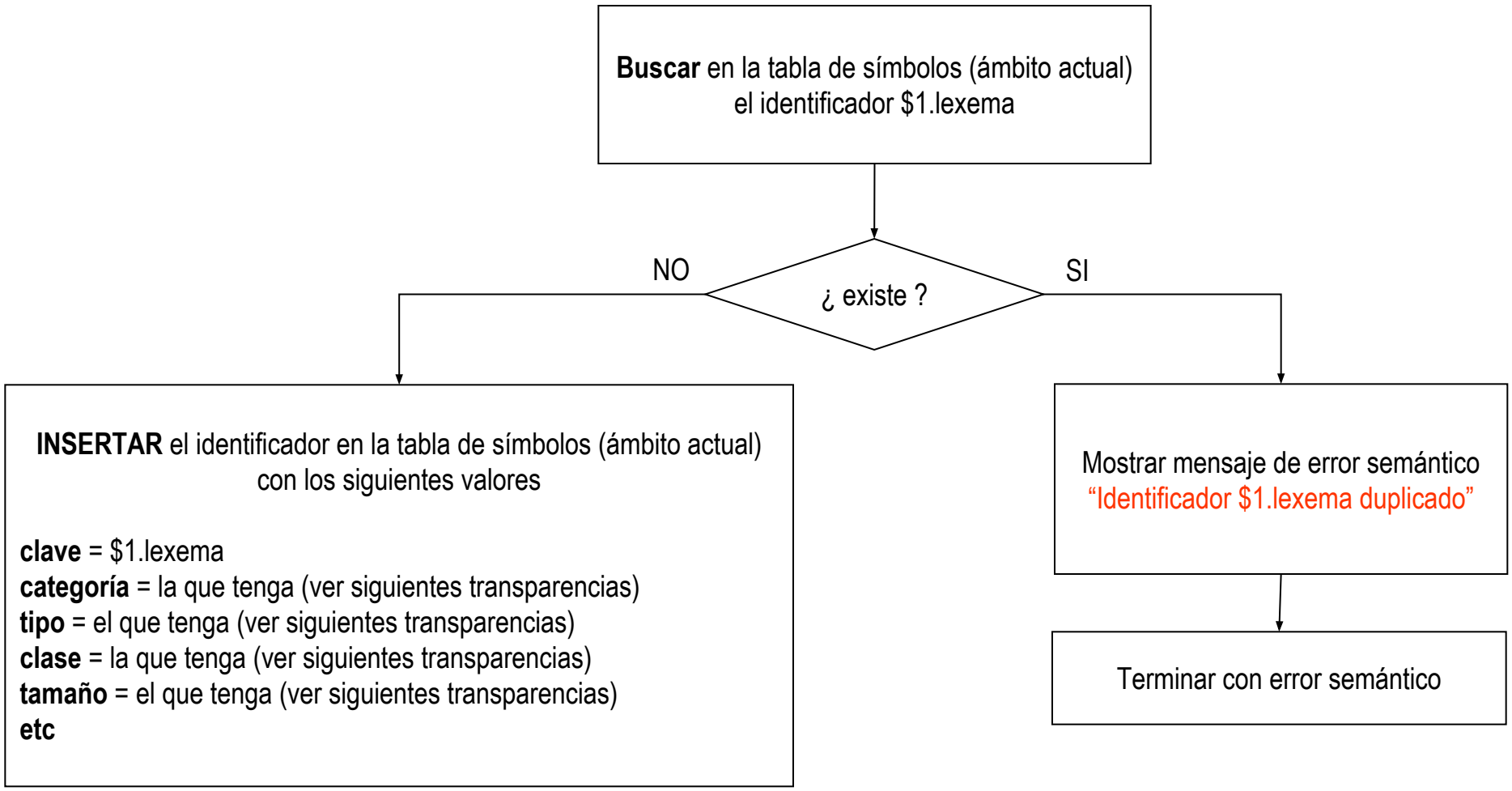
identificador : TOK_IDENTIFICADOR

- El esquema general de la acción semántica se muestra en la siguiente figura.

- La búsqueda se realiza en el ámbito donde se efectúa la inserción, es decir, el ámbito actual.

Declaración de variables globales y locales

identificador : **TOK_IDENTIFICADOR**



Propagación de las características de la declaración a las variables de la lista

- Una declaración de variables globales o locales en ÓMICRON puede ser, por ejemplo:

```
int X, Y, Z;
```

en la que se declara que las variables X, Y y Z tienen las siguientes características:

  - Son de tipo entero.
  - Son de clase escalar.

- Otra declaración podría ser:

```
array boolean [8] A1, A2;
```

en la que se declara que las variables A1 y A2 tienen las siguientes características:

  - Son de clase vector.
  - El tipo base de los vectores es el tipo lógico.
  - El tamaño de los vectores es 8.

Propagación de las características de la declaración a las variables de la lista

- **Las características/propiedades** de una declaración de **variables globales o locales** son las siguientes:
  - **categoría** (variable)
  - **clase** (escalar o vector)
  - **tipo** (entero o lógico)
  - **número de elementos** (en el caso de vectores)
  - **posición** (en el caso de variable local)

Propagación de las características de la declaración a las variables de la lista

- La producción correspondiente a una línea declarativa de variables globales o locales es la siguiente:

declaracion: modificadores_acceso clase identificadores ';'

Como puede observarse, se realiza en primer lugar la reducción del no terminal **clase** y posteriormente la reducción del no terminal **identificadores**. Por lo tanto se puede deducir que:

- Durante la reducción del no terminal clase se establecen las características de la declaración.
- El no terminal identificadores tiene que "heredar" del no terminal clase las características de la declaración.

- Debido a que la herencia de atributos no es compatible con el análisis ascendente, es necesario establecer un mecanismo para implementar dicha herencia. El mecanismo más sencillo es el uso de variables globales, una para cada atributo heredado. A continuación se describen dichas variables.

Propagación de las características de la declaración a las variables de la lista

- Una variable global, **tipo_actual**, para propagar el **tipo de la declaración**. Esta variable se actualiza con el valor correspondiente (entero o lógico) en las siguientes producciones:
  - tipo : TOK_INT
  - tipo : TOK_BOOLEAN
- Una variable global, **clase_actual**, para propagar la **clase de la declaración**. Esta variable se actualiza con el valor correspondiente (escalar o vector) en las siguientes producciones:
  - clase : clase_escalar
  - clase : clase_vector
- Una variable global, **tamanio_vector_actual** para uso exclusivo en declaraciones de vectores. Permite propagar el **tamaño** del mismo. Esta variable se actualiza en la siguiente producción:
  - clase_vector : TOK_ARRAY tipo '[' constante_entera ']'

Propagación de las características de la declaración a las variables de la lista

- Una variable global, **pos_variable_local_actual,** para propagar la **posición de una variable local** en declaraciones correspondientes a variables locales.
  Esta variable se puede gestionar de la siguiente manera:
  - se inicializa a 1 antes de la sección declarativa de una función.
  - se incrementa cada vez que se procese la declaración de una variable local, es decir, en la producción:

    identificador : TOK_IDENTIFICADOR

- En resumen, **el valor de las variables globales se establece en las acciones semánticas de las producciones adecuadas, según se ha explicado en los párrafos anteriores, con el objetivo de disponer de esos valores en las reducciones correspondientes a los identificadores que aparecen en la lista de variables de la declaración** (ver ejemplo en la transparencia 33)

Acciones semánticas de las producciones relacionadas con la clase

```
clase:    clase_escalar
              {
                    clase_actual = ESCALAR[1];
              }
          |   clase_vector
              {
                    clase_actual = VECTOR[1];
              }
               ;
```

[1] Se asume que en "omicron.h" se define:

```
#define ESCALAR  1
#define VECTOR  2
```

Acciones semánticas de las producciones relacionadas con el tipo

```
tipo:   TOK_INT
            {
                tipo_actual = INT [1];
             }
        | TOK_BOOLEAN
            {
                tipo_actual = BOOLEAN [1];
            }
            ;
```

[1] Se asume que en "omicron.h" se define:

```
#define INT  1
#define BOOLEAN  2
```

Acción semántica de la producción *clase_vector*

- En la producción *clase_vector* se puede sustituir el no terminal constante_entera por el terminal TOK_CONSTANTE_ENTERA para evitar la reducción de la regla:

```
constante_entera : TOK_CONSTANTE_ENTERA
```

```
clase_vector:   TOK_ARRAY tipo '[' TOK_CONSTANTE_ENTERA ']'
                {
                    tamanio_vector_actual = $4.valor_entero;
                    if  ((tamanio_vector_actual < 1 )  | |
                      (tamanio_vector_actual > MAX_TAMANIO_VECTOR [1]))
                    {
                            MOSTRAR MENSAJE ERROR SEMÁNTICO
                            TERMINAR CON ERROR
                    }
                }
                ;
```

[1] Se asume que en "omicron.h" se define:

```
#define MAX_TAMANIO_VECTOR  64
```

Ejemplo

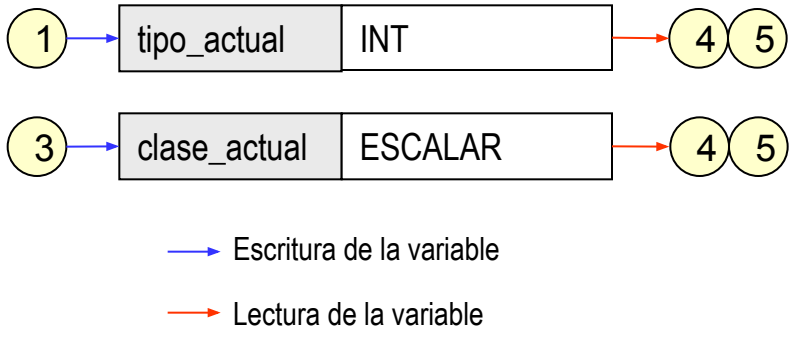
- La declaración **int X, Y;** se analiza como se muestra en la figura

PRODUCCIONES IMPLICADAS

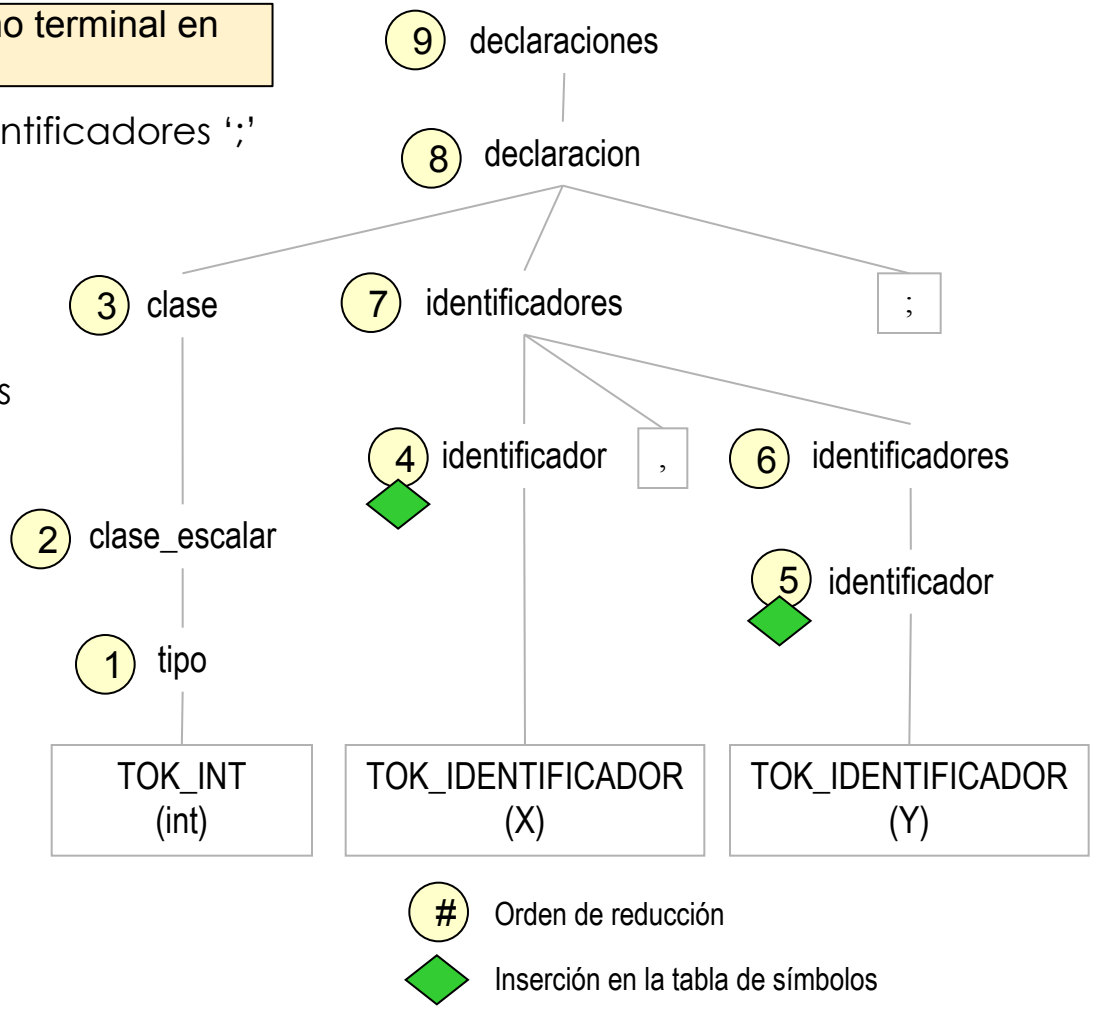
Omitimos este no terminal en el árbol

declaraciones : declaracion
declaracion : modificadores_acceso clase identificadores ';'
clase : clase_escalar
clase_escalar : tipo
tipo : TOK_INT
identificadores : identificador
identificadores : identificador ',' identificadores
identificador : TOK_IDENTIFICADOR

VARIABLES GLOBALES



ÁRBOL DE ANÁLISIS



Declaración de parámetros de función y declaración de las funciones

- La nomenclatura de los funciones derivada de la parte orientada a objetos implica que es necesario haber procesado la declaración de los parámetros antes de disponer del nombre completo de la funció (recuerda que el nombre interno tiene la estructura <clase>_<nombre_funcion>@<tipop1>...@<tipopn>
- En ese sentido, se sugiere realizar la siguientes tareas antes de declarar una función y a la hora de declarar sus parámetros
  - Acumular en alguna estructura de datos adicional los nombres de parámetros y sus tipos
  - Cuando se haya terminado la declaración de los parámetros
    - Componer el nombre de la función a partir de
      - El nombre del ámbito (clase) donde se declara
      - La información acumulada de los parámetros
- En ese instante se podrá
  - Realizar la comprobación de que la función se puede declarar
  - Insertar el símbolo de la función en el ámbito correspondiente con el nombre completo
  - Abrir el ámbito
    - Insertar el nombre de función como símbolo del nuevo ámbito
    - Iterar por la estructura de datos en la que has guardado los nombres y los tipos de los parámetros e insertarlos en la tabla de símbolos del nuevo ámbito.

Declaración de parámetros de función y declaración de las funciones

- Las reglas involucradas en la declaració de funciones son las siguientes

<funcion> → function <modificadores_acceso> <tipo_retorno> <identificador> (<parametros_funcion>)
{ <declaraciones_funcion> <sentencias> }

<tipo_retorno> → none | <tipo> | <clase_objeto>

<parametros_funcion> → <parametro_funcion> <resto_parametros_funcion> | λ

<resto_parametros_funcion> → ; <parametro_funcion> <resto_parametros_funcion> | λ

<parametro_funcion> → <tipo> <identificador> | <clase_objeto> <identificador>

<declaraciones_funcion> → <declaraciones> | λ

Declaración de parámetros de función y declaración de las funciones

- Para realizar las tareas comentadas es posible que necesites una reescritura de las reglas como se te sugiere a continuación (sólo mencionamos las nuevas)

funcion: fn_declaration sentencias '}' (1)

fn_declaration: fn_complete_name '{' declaraciones_funcion (2)

fn_complete_name: fn_name '(' parametros_funcion ')' (3)

fn_name: TOK_FUNCTION modificadores_acceso tipo_retorno TOK_IDENTIFICADOR (4)

- Esta reescritura de la gramática no introduce conflictos y te permite localizar reducciones en las que realizar las acciones
  - (1) Tareas necesarias al terminar la función. Recuerda que tal vez debas actualizar la tabla de símbolos con alguna información relacionada con el número de parámetros y de variables globales
  - (2) Tareas de inicio del cuerpo de la función
  - (3) En este punto es cuando ya se conoce el nombre de la función y la información de los parámetros para componer el nombre interno de la función
  - Recuerda que además hay cierta información global que debes mantener
    - pos_parametro_actual (que identifica inernamente, como verás, cada parámetro)
    - num_parametros_actual (esta variable almacena el número de parámetros de una función)

      Se asume que estas variables son correctamente inicializadas (consultar documentación de la gestión de identificadores de funciones)

Declaración de parámetros de función

- Para procesar la inserción de los identificadores de parámetros se diseña una nueva producción específica de la siguiente manera:

  idpf : TOK_IDENTIFICADOR

- Es necesario modificar la siguiente producción para mantener la coherencia con la nueva producción:

  parametro_funcion : tipo **idpf**

- La acción semántica de esta nueva producción se estructura así:
  - Recuerda que la comprobacion de que el identificador no exista en el ámbito actual (el de la función) y su mensaje de error, si existiera, así como el proceso de inserción en caso de no haber error hay que posponerlo a la reducción de la regla de fn_complete_name:
  - En esta regla debes acumular la información del parámetro en la estructura de datos que hayas diseñado a tal efecto. Puede resultarte cómodo un array en el que indexes esa información mediante el valor de pos_parametro_actual.
  - Recuerda que debes incrementar en uno el valor de las siguientes variables globales:
    - pos_parametro_actual
    - num_parametros_actual (esta variable almacena el número de parámetros de una función)

      Se asume que estas variables son correctamente inicializadas (consultar documentación de la gestión de identificadores de funciones)

Uso de identificadores

- Cada vez que se **usa** un identificador hay que comprobar si se ha **declarado previamente**.

- Para comprobar si un identificador ha sido declarado, simplemente se **busca en la tabla de símbolos**. Si la búsqueda termina con éxito significa que el identificador ha sido declarado y que en la compilación de su declaración se insertó en la tabla de símbolos. Es importante no olvidar que la **búsqueda se debe realizar en el ámbito actual y en todos los ámbitos que lo engloban**.

- Las producciones de la gramática que reflejan el **uso de identificadores** son aquellas pertenecientes a la **sección de sentencias** en las que aparece el no terminal "identificador".

- La producción del no terminal "identificador" ya está reservada para realizar la inserción de identificadores en la tabla de símbolos. Por lo tanto, es necesario **sustituir en las producciones de la gramática que reflejan el uso de identificadores la aparición del no terminal "identificador" por el terminal TOK_IDENTIFICADOR.**

Uso de identificadores

- Algunas de las producciones que se tienen que modificar son las siguientes:
  - asignacion : TOK_IDENTIFICADOR ‘=‘ exp
    .....
  - elemento_vector : TOK_IDENTIFICADOR ‘[‘ exp ‘]’
    .....
  - lectura : TOK_SCANF TOK_IDENTIFICADOR
    .....
  - exp : TOK_IDENTIFICADOR
    .....
  - etc

Ejemplo

```
lectura:  TOK_SCANF  TOK_IDENTIFICADOR
          {
                    Buscar en la tabla de símbolos (todos los ámbitos abiertos) el identificador  $2.lexema
                    Si no existe
                    {
                        MOSTRAR MENSAJE ERROR SEMÁNTICO
                        TERMINAR CON ERROR
                    }
             En caso contrario hacer lo correspondiente a la producción
                    {
                        COMPROBACIONES  SEMÁNTICAS
                            Si el identificador es una función   → ERROR SEMÁNTICO
                            Si el identificador es un vector  → ERROR SEMÁNTICO

                        GENERACIÓN DE CÓDIGO: APILAR LA DIRECCIÓN DEL IDENTIFICADOR
                            Si el identificador en una variable global , su dirección  es su lexema
                     Si el identificador es un parámetro o una variable local, su dirección se expresa en función de
                            ebp y la posición del  parámetro o variable local

                        GENERACIÓN DE CÓDIGO: LLAMAR A LA FUNCIÓN DE LA LIBRERÍA OLIB.O
                     Si el identificador es de tipo entero llamar a scan_int
                            Si el identificador es de tipo lógico llamar a scan_boolean
                            Restaurar la pila
                     }
          }
          ;
```

Comprobación de tipos

- **Descripción de la semántica:** En las expresiones aritméticas sólo pueden aparecer datos de tipo numérico, es decir, int. Si en una expresión aritmética aparecen operandos de distinto tipo, es decir, un entero y un booleano, se produce un error semántico.

- **Comprobaciones semánticas:** hay que comprobar que los operandos son de tipo int en las siguientes producciones y asignar el mismo tipo a la expresión resultante:

  - exp : exp '+' exp
  - exp : exp '-' exp
  - exp : exp '/' exp
  - exp : exp '*' exp
  - exp : '-' exp

Comprobaciones semánticas en las expresiones aritméticas

Implementación

- En la producciones correspondientes a expresiones aritméticas, las tareas que realiza el analizador semántico son:
  - Realizar las comprobaciones adecuadas en cada producción. Si alguna comprobación no es correcta se genera un error semántico y se termina la compilación con error.
  - Calcular (propagar) los atributos correspondientes.

- En la producción correspondiente a la suma de expresiones:

  exp : exp '+' exp

  Se comprueba que:
  - Las dos expresiones son de tipo entero, accediendo a través de $1.tipo y $3.tipo.

  Se calculan los atributos:
  - El tipo del símbolo de la parte izquierda es entero ($$.tipo = INT).
  - El símbolo de la parte izquierda no es una dirección de memoria, es un valor ($$.es_direccion =0)

Comprobaciones semánticas en las expresiones aritméticas

Implementación

- En las producciones correspondientes a las demás operaciones aritméticas se realizan las mismas comprobaciones y cálculo de atributos:
  - exp: exp '-' exp
  - exp: exp '/' exp
  - exp: exp '*' exp
  - exp: '-' exp

Comprobación de tipos

- **Descripción de la semántica:** En las expresiones lógicas sólo pueden aparecer datos de tipo boolean. El tipo de una expresión lógica boolean.

- **Comprobaciones semánticas:** hay que comprobar que los operandos son de tipo boolean en las siguientes producciones y asignar el tipo correspondiente a la expresión resultante:

  - exp : exp TOK_AND exp
  - exp : exp TOK_OR exp
  - exp : '!' exp

Comprobaciones semánticas en las expresiones lógicas

Implementación

- En las producciones correspondientes a expresiones lógicas, las tareas que realiza el analizador semántico son:
  - Realizar las comprobaciones adecuadas en cada producción. Si alguna comprobación no es correcta se genera un error semántico y se termina el proceso de compilación con error.
  - Calcular (propagar) los atributos correspondientes.

- En la producciones correspondientes a las operaciones de conjunción y disyunción de expresiones:

exp : exp TOK_AND exp
exp : exp TOK_OR exp

Se comprueba que:

- Las dos expresiones son de tipo lógico, accediendo a través de $1.tipo y $3.tipo.

Se calculan los atributos:

- El tipo del símbolo de la parte izquierda es lógico ($$.tipo = BOOLEAN).
- El símbolo de la parte izquierda no es una dirección de memoria, es un valor ($$.es_direccion =0).

Implementación

- En la producción correspondiente a la negación lógica:

  exp : '!' exp

  Sólo es necesario comprobar que el tipo de la expresión de la parte derecha es boolean.

  Se calculan los atributos:
  - El tipo del símbolo de la parte izquierda es lógico ($$.tipo = BOOLEAN).
  - El símbolo de la parte izquierda no es una dirección de memoria, es un valor ($$.es_direccion =0).

Comprobaciones semánticas en las expresiones de comparación

Comprobación de tipos

- **Descripción de la semántica:** Las comparaciones sólo pueden operar con datos de tipo numérico, siendo el resultado de la comparación de tipo boolean.

- **Comprobaciones semánticas:** hay que comprobar que los elementos comparados son de tipo numérico en las siguientes producciones y asignar el tipo correspondiente (boolean) a la expresión resultante:

  - comparacion : exp TOK_IGUAL exp
  - comparacion : exp TOK_DISTINTO exp
  - comparacion : exp TOK_MENORIGUAL exp
  - comparacion : exp TOK_MAYORIGUAL exp
  - comparacion : exp '<' exp
  - comparacion : exp '>' exp

Comprobaciones semánticas en las expresiones de comparación

Implementación

- En todas las producciones correspondientes a las comparaciones de expresiones, las únicas tareas que realiza el analizador semántico son:

  Comprobar que,
  - Las dos expresiones son de tipo numérico, accediendo a través de $1.tipo y $3.tipo.

  Calcular los atributos:
  - El tipo del símbolo de la parte izquierda es lógico ($$.tipo = BOOLEAN).
  - El símbolo de la parte izquierda no es una dirección de memoria, es un valor ($$.es_direccion =0).

- Las producciones en las que se implementan las comprobaciones semánticas son:
  - comparacion : exp TOK_IGUAL exp
  - comparacion : exp TOK_DISTINTO exp
  - comparacion : exp TOK_MENORIGUAL exp
  - comparacion : exp TOK_MAYORIGUAL exp
  - comparacion : exp '<' exp
  - comparacion : exp '>' exp

Comprobaciones semánticas para las constantes

Introducción

- Todas las producciones relacionadas con las constantes sean del tipo que sean únicamente incorporan cálculo (propagación) de atributos. La semántica de ÓMICRON no exige la realización de comprobaciones semánticas para las constantes.

- Las producciones relacionadas con las constantes son las siguientes:
  - exp: constante
  - constante : constante_logica
  - constante : constante_entera
  - constante_logica : TOK_TRUE
  - constante_logica : TOK_FALSE
  - constante_entera : TOK_CONSTANTE_ENTERA

Implementación

- En la producción que se reduce cuando el analizador morfológico identifica en la entrada una constante entera, se realiza la siguiente propagación de atributos:

```
constante_entera: TOK_CONSTANTE_ENTERA
        {
                $$.tipo = INT;
        $$.es_direccion = 0;
        }
        ;
```

Comprobaciones semánticas para las constantes

Implementación

- Las producciones que se reducen cuando el analizador morfológico identifica en la entrada una constante booleana, contienen la siguiente propagación de atributos:

```
constante_logica: TOK_TRUE
        {
                $$.tipo = BOOLEAN;
                $$.es_direccion = 0;
        }
        ;
constante_logica: TOK_FALSE
        {
                $$.tipo = BOOLEAN;
                $$.es_direccion = 0;
        }
        ;
```

Implementación

- Por último, las producciones intermedias que representan que una constante es entera o lógica, tienen el siguiente cálculo de atributos:

```
constante : constante_logica
   {
      $$.tipo = $1.tipo;
      $$.es_direccion = $1.es_direccion;
   }
constante : constante_entera
   {
      $$.tipo = $1.tipo;
      $$.es_direccion = $1.es_direccion;
   }
   ;
```

Comprobaciones semánticas para otras expresiones

Introducción

- El no terminal **exp** tiene varias producciones además de las relativas a expresiones aritméticas y lógicas cuyas comprobaciones semánticas ya han sido descritas. Estas producciones son:
  - exp: TOK_IDENTIFICADOR
  - exp: constante
  - exp: '(' exp ')'
  - exp: '(' comparacion ')'
  - exp: elemento_vector
  - exp: TOK_IDENTIFICADOR '(' lista_expresiones ')'
- Las comprobaciones semánticas de cada una de las producciones se describe a continuación.

Comprobaciones semánticas para otras expresiones

Implementación

- En la producción siguiente:

exp : TOK_IDENTIFICADOR

Se comprueba que:

- El identificador ($1.lexema) existe en la tabla de símbolos (en cualquiera de los ámbitos abiertos). Si el identificador no existe, se genera un error semántico y se termina el proceso de compilación con error.
- El identificador no es ni de categoría función (es decir, es una variable o un parámetro), ni de clase vector (es decir, es un escalar).

Se calculan los atributos:

- El tipo del símbolo exp de la parte izquierda ($$.tipo) se establece haciendo uso de la información almacenada en la tabla de símbolos para el identificador de la parte derecha de la producción. (IMPORTANTE, $1.tipo no almacena el tipo del identificador, el tipo está en la tabla de símbolos).
- El símbolo de la parte izquierda representa una dirección de memoria ($$.es_direccion=1).

Comprobaciones semánticas para otras expresiones

Implementación

- Las producciones de las diapositivas anteriores únicamente contienen en su acción semántica las siguientes instrucciones para propagar atributos:
  - $$.tipo = $1.tipo en las producciones exp: constante y exp: elemento_vector.
  - $$.tipo = $2.tipo en las producciones exp: ‘(‘ exp ‘)’ y exp: ‘(‘ comparacion ‘)’.
  - $$.es_direccion = $1.es_direccion en las producciones constante y exp: elemento_vector.
  - $$.es_direccion = $2.es_direccion en las producciones ‘(‘ exp ‘)’ y exp: ‘(‘ comparacion ‘)’.

Comprobaciones semánticas para otras expresiones

Implementación

- La siguiente producción define la sintaxis de la llamada a una función:

  exp: TOK_IDENTIFICADOR '(' lista_expresiones ')'

- La descripción de la semántica es la siguiente:
  - Como se verá la comprobación de la corrección de los tipos de los argumentos y de su número se realiza gracias a la nomenclatura de las funciones
  - En las llamadas a funciones, no se puede escribir, en un parámetro, una llamada a otra función.

- Hay que añadir un mecanismo que, durante la reducción de lista_expresiones, permita determinar el sufijo adecuado al nombre de la función.
- Recuerda que el nombre de la función debe componerse de la siguiente manera
  - El nombre de la clase / ámbito main en el que se declaró
  - El nombre de alto nivel de la función
  - El sufijo que informa de los tipos de sus argumentos
- Una vez compuesto el nombre, se comprueba que el identificador existe en la tabla de símbolos y además tiene categoría de función.

Implementación

- Para componer el sufijo, se puede articular un mecanismo similar al que se describió para el nombre de función en su declaración:
  - Diseñar una estructura que acumule los tipos de las expresiones de los argumentos que se sintetiza al reducirlas.
  - Puedes indexar esa información utilizando la variable global num_parametros_llamada_actual o cualquier otra que decidas a tal efecto

- Un posible mecanismo sería usar una variable global **num_parametros_llamada_actual** para contar el número de parámetros de la llamada. La inicialización a 0 de esta variable se haría antes de comenzar a procesar el no terminal lista_expresiones, y se incrementaría en 1 cada vez que se procesará uno de los elementos de la lista.

- Para comprobar que no haya una llamada a función en el lugar de un parámetro, se puede utilizar otra variable global **en_explist** que se inicialice a 1 antes de procesar el no terminal lista_expresiones. Y posteriormente, en cualquier llamada a función se comprobará que en_explist no sea 1. Por último, al terminar la llamada a función se reseteará a 0 esta variable.

Implementación

- Agrupando todas las tareas necesarias para implementar la semántica de la llamadas a funciones, se deduce que hay dos puntos en los que se ubican las acciones:

  exp: TOK_IDENTIFICADOR ❶ '(' lista_expresiones ')' ❷

- Se modifica la gramática para situar la acción del punto ❶ al final de una producción, añadiendo la siguiente regla:

  idf_llamada_funcion : TOK_IDENTIFICADOR ❶

  Por lo tanto, la producción 88 de la gramática original ahora quedaría de la siguiente manera:

  exp: idf_llamada_funcion '(' lista_expresiones ')' ❷

Comprobaciones semánticas para otras expresiones

Implementación

- La acción del punto ❶ implica las siguientes tareas:
  - Como después del identificador empieza la lista de expresiones correspondiente a los parámetros de la llamada, hay que comprobar que la variable en_explist no valga 1 para asegurar que la actual llamada a función no es un parámetro de llamada a función.
  - Inicializar a 0 la variable num_parametros_llamada_actual.
  - Inicializar a 1 la variable en_explist ya que después del identificador empieza la lista de expresiones correspondiente a los parámetros de la llamada.
  - Propagar el lexema del identificador ya que se va a necesitar en el punto ❷.

- La acción del punto ❷ implica las siguientes tareas:
  - Puedes componer el nombre interno de la función a partir del nombre de la clase / ámbito main en el que se ha declarado y la información acumulada del tipo de los argumentos en el proceso de lista de expresiones que será utilizado para construir el sufijo
  - Buscar el identificador en la tabla de símbolos. Generar un error semántico si no existe y terminar el proceso de compilación con error.
  - Comprobar que el identificador es de categoría función.
  - La comprobación de la corrección del número y posición de los tipos de los argumentos con los de la declaración se realiza automáticamente dado que está codificado en el nombre buscado
  - Resetear a 0 la variable en_explist.
  - El tipo del símbolo de la parte izquierda exp ($$.tipo) se iguala al tipo de retorno de la función que está almacenado en la tabla de símbolos.
  - El símbolo de la parte izquierda no representa una dirección, es un valor ($$.es_direccion=0).

Implementación

- Por último, sólo faltaría decidir los puntos en los que se debe actualizar la variable global num_parametros_llamada_actual. Las producciones indicadas para ello son las siguientes:

  (1) lista_expresiones: exp resto_lista_expresiones
  (2) resto_lista_expresiones: ',' exp resto_lista_expresiones

  En ambas producciones únicamente hay que realizar la acción:

  num_parametros_llamada_actual++

- Es también en esta regla (tras reducir exp donde dispondrás en $1.tipo (1) y en $2.tipo (2) el tipo que te permitirá guardar la información de tipo de ese argumento para el sufijo del nombre de función

Comprobación de tipos

- **Descripción de la semántica:** Para que una sentencia de asignación sea válida, los tipos de dato de la parte izquierda y derecha deben ser iguales.

- **Comprobaciones semánticas:** hay que comprobar que son iguales los tipos de la parte izquierda y derecha de la asignación en las siguientes producciones:

  - asignacion : TOK_IDENTIFICADOR '=' exp
  - asignacion : elemento_vector '=' exp

Implementación

- La primera producción corresponde a la asignación de una expresión a un identificador:

  asignacion: TOK_IDENTIFICADOR '=' exp

  Se comprueba que:
  - El identificador existe en la tabla de símbolos.
  - El identificador no es de categoría función (es decir, es una variable o un parámetro) y no es de clase vector (es decir, es un escalar).
  - Los tipos de ambas partes de la asignación son iguales.

  No hay propagación de atributos, no es necesario.

- La segunda producción corresponde a la asignación de una expresión a un elemento de un vector:

  asignacion: elemento_vector '=' exp

  Se comprueba que:
  - Los tipos de ambas partes de la asignación son iguales.

  No hay propagación de atributos, no es necesario.

Comprobaciones semánticas para vectores

Comprobación del tamaño

- **Descripción de la semántica:** El tamaño de los vectores no podrá exceder nunca el valor de 64.

- **Comprobaciones semánticas:** hay que comprobar que el tamaño declarado para un vector no excede el límite permitido (64). También hay que comprobar que el tamaño declarado es mayor que 0 (esta restricción no se especifica explícitamente en la descripción de la semántica del lenguaje). Estas comprobaciones se pueden realizar la siguiente producción:

  - clase_vector : TOK_ARRAY tipo '[' TOK_CONSTANTE_ENTERA ']'

(VER EPÍGRAFE "Acciones semánticas de la producción *clase_vector*")

Comprobaciones en el indexado de vectores

- **Descripción de la semántica:** Para indexar los elementos de un vector se utiliza la cadena "[exp]"a continuación del nombre del vector. Los corchetes deberán contener en su interior una expresión de tipo **int**.

- **Comprobaciones semánticas:** la producción indicada para comprobar la semántica descrita es la siguiente:

  - elemento_vector : TOK_IDENTIFICADOR '[' exp ']'

  Las comprobaciones que hay que realizar son:

  - El identificador que aparece ($1.lexema) existe en la tabla de símbolos y además corresponde a la declaración de un vector.
  - La expresión que actúa como índice es de tipo INT ($3.tipo == INT).

  Se propagan los atributos:

  - $$.tipo = el tipo que aparece para el lexema $1.lexema en la tabla de símbolos.
  - $$.es_direccion = 1.

Comprobaciones en el indexado de vectores de una dimensión

- **Descripción de la semántica:** Los corchetes deberán contener en su interior una expresión de tipo **int**, con un valor entre 0 y el tamaño definido para el vector menos 1 (ambos incluidos).

- **Comprobaciones semánticas:** en una operación de indexado de un vector de una dimensión, la comprobación de que el valor del índice está dentro del rango permitido sólo se puede realizar en **tiempo de ejecución**, no en tiempo de compilación. Para implementar esta restricción semántica se genera el código ensamblador adecuado para realizar la validación del valor del índice en tiempo de ejecución. La producción indicada para la generación del código es:

  - elemento_vector : TOK_IDENTIFICADOR '[' exp ']'

Comprobaciones semánticas para las condiciones

- **Descripción de la semántica:** Las expresiones que aparecen como condiciones en las sentencias **if**, **if-else** y **while** deben ser de tipo boolean.

- **Comprobaciones semánticas:** hay que comprobar que las condiciones de los bucles while y de las sentencias condicionales son de tipo boolean.
  Las producciones implicadas en la comprobación de esta restricción semántica son las siguientes:

  - condicional : TOK_IF '(' exp ')' '{' sentencias '}'
  - condicional : TOK_IF '(' exp ')' '{' sentencias '}' TOK_ELSE '{' sentencias '}'
  - bucle : TOK_WHILE '(' exp ')' '{' sentencias '}'

  Las comprobaciones se realizan sobre el no terminal exp que es el que representa la condición.

Comprobaciones semánticas para las condiciones

- **Modificación de la gramática:** los puntos adecuados para realizar las comprobaciones sobre el símbolo exp se sitúan después de él y antes del no terminal sentencias. Por ejemplo, para la sentencia condicional simple, los puntos posibles son los indicados con **{acción}** :
  - condicional : TOK_IF '(' exp **{acción}** ')' '{' sentencias '}'
  - condicional : TOK_IF '(' exp ')' **{acción}** '{' sentencias '}'
  - condicional : TOK_IF '(' exp ')' '{' **{acción}** sentencias '}'

Una vez elegido el punto, para poder situar la acción semántica al final de la regla, sería necesario modificar la gramática. Por ejemplo si se elige el primer punto, la gramática se manipula de la siguiente manera:

- Se añade un nuevo símbolo no terminal **if_exp** para situar la acción semántica al final de una regla:

  if_exp : TOK_IF '(' exp **{acción}**

- La producción original quedaría de la siguiente manera:

  condicional : if_exp ')' '{' sentencias '}'

Comprobaciones semánticas para las condiciones

- En la nueva regla,

  if_exp : TOK_IF '(' exp **{acción}**

  Se comprueba que:
  - La expresión es de tipo boolean, es decir, $3.tipo == BOOLEAN

- Se aplica el mismo razonamiento para la sentencias **if-else** y **while**.

Comprobaciones para sentencias de entrada

- **Descripción de la semántica:** Las operación de entrada **scanf** se efectúa sobre elementos de clase escalar que son identificadores.

- **Comprobaciones semánticas:** La producción de entrada de datos es:

  - lectura : TOK_SCANF TOK_IDENTIFICADOR

  Solamente es necesario realizar la siguiente comprobación:

  - El identificador ($2.lexema) existe en la tabla de símbolos, y es de clase escalar.

Comprobaciones para sentencias de salida

- **Descripción de la semántica:** La operación de escritura **printf** trabaja con expresiones.

- **Comprobaciones semánticas:** La producción correspondiente a la operación de escritura **printf** es la siguiente:
  - escritura : TOK_PRINTF exp

    En esta producción no hay que realizar ninguna comprobación semántica.

Otras comprobaciones semánticas

- **En la descripción de la semántica de ÓMICRON proporcionada en el enunciado de la práctica sólo se mencionan los aspectos más relevantes. Además de éstos, hay un conjunto de restricciones semánticas que también deben estar presentes en el compilador de ÓMICRON, como por ejemplo:**
  - Una sentencia de retorno de función solamente debe aparecer en el cuerpo de una función.
  - En el cuerpo de una función obligatoriamente tiene que aparecer al menos una sentencia de retorno.
  - En una sentencia de retorno el tipo de la expresión debe de coincidir con el tipo de retorno de la función.
  - etc