Índice

Marina de la Cruz
Alfonso Ortega

Índice

Índice

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.

Generación de código correspondiente a la declaración de variables globales

Ubicación de la acción

- **Cuando termina la compilación de la sección de declaración de variables globales de un programa ÓMICRON, todas las variables globales declaradas están almacenadas en la tabla de símbolos. En ese punto es posible generar el código ensamblador correspondiente a la declaración de las variables (segmento de datos)** Por lo tanto, el punto adecuado para escribir el segmento de datos del fichero ensamblador es el indicado por el símbolo en la producción del axioma:

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

  Se crea un nuevo no terminal y se añade una producción lambda para dicho símbolo cuya acción semántica sea **generar el código ensamblador correspondiente a la declaración de las variables** . Por ejemplo:

```
programa: TOK_MAIN '{' declaraciones escritura_TS funciones sentencias '}'
        ;
escritura_TS: {
          /* Tarea adecuada */
        }
        ;
```

- Esta acción semántica se puede "aprovechar" para escribir la cabecera del segmento de código, que es un texto fijo.

Generación de código correspondiente a la declaración de variables globales

Resultado esperado de la acción

Programa ÓMICRON

```
main {
  int x;
  boolean a;
  array int [8] v8;
  array boolean [3] u3;
  …
```

Programa NASM

```
segment .bss
  _x resd 1
  _a resd 1
  _v8 resd 8
  _u3 resd 3
...
```

Procesado de la sección declarativa

Reducción de la nueva regla escritura_TS

Tabla de símbolos

clave	categoría	tipo	clase	tamaño	etc
x	variable	INT	ESCALAR	-	
a	variable	BOOLEAN	ESCALAR	-	
v8	variable	INT	VECTOR	8	
u3	variable	BOOLEAN	VECTOR	3	

Generación de código correspondiente a la declaración de variables globales

Esquema de la acción

- Según lo visto anteriormente, la acción semántica asociada a la sección declarativa contendría:
  - La escritura del segmento de datos bss.
  - La escritura de la cabecera del segmento de código.

- Por razones de estructuración, se sugiere escribir una librería que contenga funciones de apoyo a la generación de código. De momento esta librería contendría:
  - Una función que a partir de la tabla de símbolos escriba el segmento de datos correspondiente.
  - Una función que escriba la cabecera del segmento de código, que es como sigue:

```
segment .text
global main
extern scan_int,  scan_boolean
extern print_int,  print_boolean, print_string, print_blank, print_endofline
```

PENDIENTE 1: escritura de segmento .data

Generación de la etiqueta "main:"

- La etiqueta "main:" tiene que ser escrita en el fichero **antes de la primera sentencia** correspondiente al programa principal. Por lo tanto, el punto adecuado es el indicado por el símbolo en la producción del axioma:

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

  Se crea un nuevo no terminal y se añade una producción lambda para dicho símbolo cuya acción semántica sea **escribir en el fichero ensamblador la etiqueta "main:"**. Por ejemplo:

```
programa: TOK_MAIN '{' declaraciones escritura_TS funciones escritura_main sentencias '}'
          ;
escritura_main: {
                    /* Tarea adecuada */
                }
                ;
```

Generación de la etiqueta “main:”

- Se puede añadir a la librería que contiene las funciones de apoyo a la generación de código una función que simplemente escriba la etiqueta “main:” en el fichero ensamblador.

- Posteriormente se estudiará en profundidad las tareas que debe realizar el generador de código en relación al no terminal “funciones”.

PENDIENTE 2: salvar registros como primera acción del programa ensamblador.

Contenido del fichero ensamblador antes de la traducción de la primera sentencia

- Según lo visto anteriormente, el contenido del fichero ensamblador antes de la traducción de primera sentencia sería el siguiente:

```
segment .data
;  declaración de variables inicializadas (ver transparencias siguientes)
;
segment .bss
;  declaración de variables según el contenido de la tabla de símbolos
;
segment .text
global main
extern scan_int,  scan_boolean
extern print_int,  print_boolean, print_string, print_blank, print_endofline
;
;  código correspondiente a la compilación del no terminal "funciones"
;
main:
```

Control de errores en tiempo de ejecución

- La gestión de errores en tiempo de ejecución requiere que en el proceso de compilación se genere el código ensamblador que comprueba dichos errores.

- El código ensamblador asociado a un control de error puede funcionar de la siguiente manera:
  - comprueba la condición de error
  - si se cumple la condición de error, se transfiere el flujo de ejecución a una etiqueta en la que se realizan dos acciones:
    - se informa al usuario del error ocurrido
    - se salta de manera incondicional al final del programa

- Para poder informar de los errores, se tienen que declarar los mensajes en el segmento de datos inicialiados .data, por ejemplo:

```
segment   .data
mensaje_1   db   "Indice fuera de rango" , 0
mensaje_2   db   "División por cero" , 0
.....
```

- En la parte final del programa se puede dedicar una zona para las etiquetas en las que se informa al usuario de un error de ejecución. La estructura de esta zona sería la siguiente:

```
error_1:   push dword mensaje_1
           call print_string
           add esp, 4
           jmp near fin
error_2:   push dword mensaje_2
           call print_string
           add esp, 4
           jmp near fin
fin: ret
```

NOTA: la estructura del código ensamblador podría ser diferente.

- Obviamente la producción en la que se puede escribir la zona final del programa ensamblador es la regla del axioma.

Aclaraciones previas a la generación de código de sentencias

- El compilador de ÓMICRON genera código ensamblador para una **máquina a pila**.
- Cuando sea necesario apilar una **variable**, se apilará siempre su **dirección**, no su contenido (en NASM, la dirección de una variable es el nombre de la misma)
- Cuando se apile una **constante**, se apilará su **valor**.
- La generación de código implica la creación automática de etiquetas (en sentencias condicionales, bucles de control, comparaciones, etc)

  Por lo tanto, es necesario un mecanismo que asegure que las **etiquetas** que se utilicen en el programa ensamblador son **únicas**.

  Un posible mecanismo es usar una variable entera global (etiqueta), inicializada a un valor concreto, por ejemplo 1, y que sea incrementada cada vez que se utilice en la generación de código.

Generación de código para las expresiones de tipo identificador

- La producción de la gramática correspondiente a las expresiones de tipo identificador es la siguiente:

  exp : TOK_IDENTIFICADOR

- En general, la generación de código que se realiza en esta producción consiste en **apilar la dirección del identificador**. Por lo tanto la evolución de la pila gráficamente sería:

Generación de código para las expresiones de tipo identificador

- El acceso a la dirección del identificador es diferente dependiendo de que el identificador sea:
  - Variable global
  - Variable local
  - Parámetro de función
- Por lo tanto es necesario un mecanismo que permita saber si una variable es local o global.
- Además, si el identificador corresponde a un parámetro de una llamada a función, es necesario realizar las acciones específicas que permitan dejar en la pila el valor del identificador en lugar de su dirección (los parámetros se pasan por valor)
- Si el identificador corresponde a una **variable global**, el código que se debe generar consiste en apilar la dirección del identificador, que se corresponde con su nombre.
- Los demás casos los estudiaremos posteriormente, después de conocer la generación de código para funciones.

PENDIENTE 3: expresiones de tipo identificador que no son variables globales.

Introducción

- El **objetivo** de la generación de código para las constantes es **apilar el valor de la constante**. Hay distintas alternativas para conseguir este objetivo.

- Estudiaremos los dos casos:
  - Constantes enteras
  - Constantes lógicas

Generación de código para las constantes

Constantes lógicas

- Las producciones de la gramática en las que se realiza la inserción en la pila de la constante lógica son las siguientes:
  - constante_logica: TOK_TRUE
  - constante_logica: TOK_FALSE
- Por ejemplo, la acción semántica de la primera producción, incorporando análisis semántico y generación de código, podría ser:

```
constante_logica:  TOK_TRUE
                {
                    /*  análisis semántico */
                    $$.tipo = BOOLEAN;
                    $$.es_direccion = 0;

                    /* generación de código */
                    fprintf(fichero_ensamblador, "; numero_linea %d\n", numero_linea);
                    fprintf(fichero_ensamblador, "\tpush dword 1\n");
                }
                ;
```

Generación de código para las constantes

Constantes enteras

- La producción de la gramática en la que se realiza la inserción en la pila de la constante entera es la siguiente:
  - constante_entera: TOK_CONSTANTE_ENTERA
- La acción semántica de la producción, incorporando análisis semántico y generación de código, podría ser:

```
constante_entera:  TOK_CONSTANTE_ENTERA
                {
                    /*  análisis semántico */
                    $$.tipo = INT;
                    $$.es_direccion = 0;

                    /* generación de código */
                    fprintf(fichero_ensamblador, "; numero_linea %d\n", numero_linea);
                    fprintf(fichero_ensamblador, "\tpush dword %d\n", $1.valor_entero);
                }
                ;
```

Introducción

- El código NASM que el compilador tiene que generar para la ejecución de una operación aritmética sigue la siguiente secuencia:
  - Antes de realizar cualquier operación aritmética, los operandos están en la pila.
  - Se extraen los operandos de la pila a los registros.
  - Se realiza la operación aritmética con el código de operación adecuado.
  - Se deposita en la pila el resultado de la operación.

- Los **registros** que se utilizan para aritmética entera, son los de **propósito general**: eax, edx, etc.

- Los códigos de operación son, entre otros: add, imul, sub, etc. Y se detallarán más adelante.

Reglas de la gramática para las operaciones aritméticas

- Las reglas de la gramática relativas a las operaciones aritméticas son:
  - exp : exp '+' exp
  - exp : exp '-' exp
  - exp : exp '/' exp
  - exp : exp '*' exp
  - exp : '-' exp

- La generación de código de todas las reglas anteriores comparten la misma estructura básica. A continuación se muestra el caso de la suma.

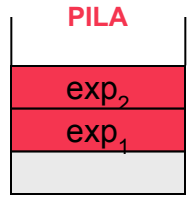
Generación de código para la operación suma

$$exp : exp_1 \text{ ‘+’ } exp_2$$

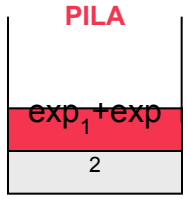
GENERACIÓN DE CÓDIGO

```
; cargar el segundo operando en edx
pop dword edx
SI ($3.es_direccion == 1)
  mov dword edx , [edx]

; cargar el primer operando en eax
pop dword eax
SI ($3.es_direccion == 1)
  mov dword eax , [eax]

; realizar la suma y dejar el resultado en eax
add eax, edx

; apilar el resultado
push dword eax
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN



DESPUÉS DE LA REDUCCIÓN

Generación de código para la operación suma

- La generación de código se puede ubicar directamente en la acción semántica de la regla, o bien programando una función. Para la suma de expresiones de tipo entero, la función tendría el siguiente aspecto:

```
void gc_suma_enteros(FILE* f, es_direccion_op1, es_direccion_op2)
{
     fprintf(f, "; cargar el segundo operando en edx\n");
     fprintf(f, "pop dword edx\n");
     if (es_direccion_op2 == 1)
    fprintf(f, "mov dword edx , [edx]\n");
     fprintf(f, "; cargar el primer operando en eax\n");
fprintf(f,"pop dword eax\n");
        if (es_direccion_op1 == 1)
fprintf(f, "mov dword eax , [eax]\n");
     fprintf(f, "; realizar la suma y dejar el resultado en eax \n");
     fprintf(f, "add eax,edx\n);
     fprintf(f, "; apilar el resultado\n");
     fprintf(f, "push dword eax\n");
     return;
}
```

Generación de código para el resto de las operaciones aritméticas binarias

- Las producciones correspondientes al resto de las operaciones aritméticas binarias son las siguientes:
  - exp : exp '-' exp
  - exp : exp '/' exp
  - exp : exp '*' exp

- El código ensamblador correspondiente a las producciones anteriores es similar al de la regla de la suma visto anteriormente. La única diferencia es la instrucción ensamblador que se utilice y las posibles peculiaridades de la misma.

- Para la **resta** se puede utilizar la instrucción ensamblador siguiente:

```
sub eax,edx
```

que resta al contenido del registro `eax` el contenido del registro `edx` y deja el resultado en `eax`.

Generación de código para el resto de las operaciones aritméticas binarias

- Para la **división** se puede utilizar la instrucción ensamblador de división entera **`idiv`**. Esta instrucción trabaja de la siguiente manera:
  - Asume que el dividendo está en la composición de registros `edx:eax`
  - El divisor es el registro que aparece en la instrucción.
  - El resultado de la división entera se ubica en el registro `eax`

- Para que el dividendo esté en la composición de registros `edx:eax` y realizar correctamente la división se tiene que hacer:
  - Cargar el divisor en `ecx`
  - Cargar el dividendo en `eax`
  - Extender el dividendo a la composición de registros `edx:eax` (`cdq`)
  - Realizar la división (`idiv ecx`)

- Se debe controlar la división por cero según se explicó en el apartado "Control de errores en tiempo de ejecución"

Generación de código para el resto de las operaciones aritméticas binarias

- Para el **producto** se puede utilizar la instrucción ensamblador de multiplicación entera **`imul`**. Esta instrucción asume que uno de los operandos está en el registro `eax`, y el otro operando está en el registro que aparece en la instrucción. El resultado se carga en `edx:eax`, y consideraremos el valor de `eax` como resultado de la operación.

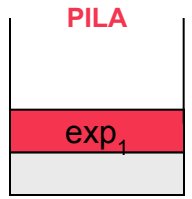
Generación de código para la negación de un operando

exp : '-' exp_1

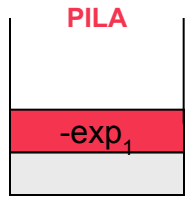
GENERACIÓN DE CÓDIGO

```
; cargar el operando en eax
pop dword eax
SI ($2.es_direccion == 1)
   mov dword eax , [eax]

; realizar la negación. El resultado en eax
neg eax

; apilar el resultado
push dword eax
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN



DESPUÉS DE LA REDUCCIÓN

Generación de código para las operaciones lógicas

Introducción

- Las producciones de la gramática correspondientes a las operaciones lógicas son las siguientes:
  - exp : exp TOK_AND exp
  - exp : exp TOK_OR exp
  - exp : '!' exp
- De la misma manera que en las operaciones aritméticas, cuando se reduce una de las anteriores producciones los operandos están en la pila porque han sido reducidos previamente por alguna de las producciones del no terminal "exp".
- El código ensamblador correspondiente a operaciones lógicas usa la pila con la siguiente secuencia de acciones:
  - Desapilar los operandos.
  - Realizar la operación lógica correspondiente.
  - Apilar el resultado de la operación.

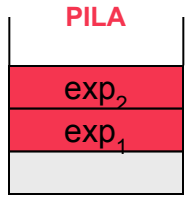
Generación de código para la conjunción lógica

exp : exp_1 **TOK_AND** exp_2

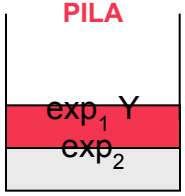
GENERACIÓN DE CÓDIGO

```
; cargar el segundo operando en edx
pop dword edx
SI ($3.es_direccion == 1)
  mov dword edx , [edx]

; cargar el primer operando en eax
pop dword eax
SI ($1.es_direccion == 1)
  mov dword eax , [eax]

; realizar la conjunción y dejar el resultado en eax
and eax , edx

; apilar el resultado
push dword eax
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN



DESPUÉS DE LA REDUCCIÓN

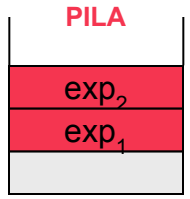
Generación de código para la disyunción lógica

exp : exp_1 **TOK_OR** exp_2

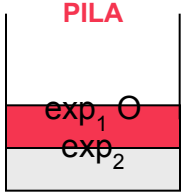
GENERACIÓN DE CÓDIGO

```
; cargar el segundo operando en edx
pop dword edx
SI ($3.es_direccion == 1)
  mov dword edx , [edx]

; cargar el primer operando en eax
pop dword eax
SI ($1.es_direccion == 1)
  mov dword eax , [eax]

; realizar la conjunción y dejar el resultado en eax
or eax , edx

; apilar el resultado
push dword eax
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN



DESPUÉS DE LA REDUCCIÓN

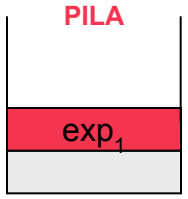
Generación de código para la negación lógica

exp : **'!'** exp_1

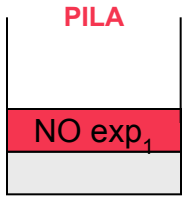
GENERACIÓN DE CÓDIGO

```
; cargar el operando en eax
pop dword eax
SI ($2.es_direccion == 1)
   mov dword eax , [eax]

; ver si eax es 0 y en ese caso saltar a negar_falso
or eax , eax
jz near negar_falso#

; cargar 0 en eax (negación de verdadero) y saltar al final
mov dword eax,0
jmp near fin_negacion#

; cargar 1 en eax (negación de falso)
negar_falso#:     mov dword eax,1

; apilar eax
fin_negacion#:    push dword eax
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN



DESPUÉS DE LA REDUCCIÓN

UNICIDAD DE ETIQUETAS

etiqueta ++

Generación de código para las comparaciones

Introducción

- La **secuencia** para la ejecución de una comparación es la siguiente:
  - Los operandos de la comparación están en la pila.
  - Se extraen los operandos de la pila a los registros adecuados.
  - Se realiza la comparación con el código de operación adecuado.
  - Se deposita en la pila el resultado de la comparación (1 si ha terminado con éxito y 0 en el caso contrario)

- Las producciones de la gramática correspondientes a las comparaciones son las siguientes:
  - 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

Generación de código para las comparaciones

- En general, la implementación en ensamblador de una comparación se hará de la siguiente manera:
  - Suponiendo que los operandos están en los registros eax y edx
  - Se utiliza la instrucción:

    cmp eax , edx

  - Dependiendo del resultado de la comparación se redirige el flujo de ejecución mediante **saltos condicionales**. Las instrucciones de salto disponibles, para comparaciones de enteros con signo) son las siguientes:

    je salta si eax == edx (también **jz**)
    jne salta si eax != edx (también **jnz**)
    jle salta si eax <= edx (también **jng**)
    jge salta si eax >= edx (también **jnl**)
    jl salta si eax < edx (también **jnge**)
    jg salta si eax > edx (también **jnle**)

  - En los destinos de los saltos se procederá a apilar un 0 o un 1 dependiendo del éxito de la comparación.

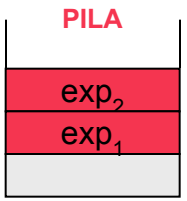
Generación de código para la comparación "=="

comparacion : exp_1 **TOK_IGUAL** exp_2

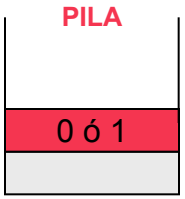
GENERACIÓN DE CÓDIGO

```
; cargar la segunda expresión en edx
pop dword edx
SI ($3.es_direccion == 1)
  mov dword edx , [edx]

; cargar la primera expresión en eax
pop dword eax
SI ($1.es_direccion == 1)
  mov dword eax , [eax]

; comparar y apilar el resultado
        cmp eax, edx
        je near igual#
        push dword 0
        jmp near fin_igual#
igual#: push dword 1
fin_igual#:
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN



DESPUÉS DE LA REDUCCIÓN

UNICIDAD DE ETIQUETAS

etiqueta ++

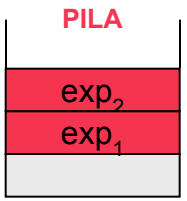
Generación de código para la comparación "!="

comparacion : exp_1 **TOK_DISTINTO** exp_2

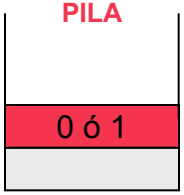
GENERACIÓN DE CÓDIGO

```
; cargar la segunda expresión en edx
pop dword edx
SI ($3.es_direccion == 1)
  mov dword edx , [edx]

; cargar la primera expresión en eax
pop dword eax
SI ($1.es_direccion == 1)
  mov dword eax , [eax]

; comparar y apilar el resultado
        cmp eax, edx
        jne near distinto#
        push dword 0
        jmp near fin_distinto#
distinto#:      push dword 1
fin_igual#:
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN



DESPUÉS DE LA REDUCCIÓN

UNICIDAD DE ETIQUETAS

etiqueta ++

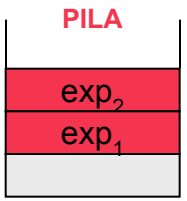
Generación de código para la comparación "<="

comparacion : exp_1 **TOK_MENORIGUAL** exp_2

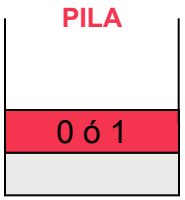
GENERACIÓN DE CÓDIGO

```
; cargar la segunda expresión en edx
pop dword edx
SI ($3.es_direccion == 1)
  mov dword edx , [edx]

; cargar la primera expresión en eax
pop dword eax
SI ($1.es_direccion == 1)
  mov dword eax , [eax]

; comparar y apilar el resultado
                cmp eax, edx
                jle near menorigual#
                push dword 0
                jmp near fin_menorigual#
menorigual#:        push dword 1
fin_menorigual#:
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN



DESPUÉS DE LA REDUCCIÓN

UNICIDAD DE ETIQUETAS

etiqueta ++

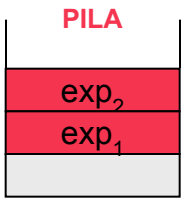
Generación de código para la comparación ">="

comparacion : exp_1 **TOK_MAYORIGUAL** exp_2

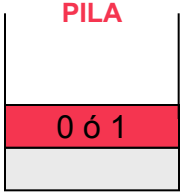
GENERACIÓN DE CÓDIGO

```
; cargar la segunda expresión en edx
pop dword edx
SI ($3.es_direccion == 1)
  mov dword edx , [edx]

; cargar la primera expresión en eax
pop dword eax
SI ($1.es_direccion == 1)
  mov dword eax , [eax]

; comparar y apilar el resultado
            cmp eax, edx
            jge near mayorigual#
            push dword 0
            jmp near fin_mayorigual#
mayorigual#:      push dword 1
fin_mayorigual#:
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN



DESPUÉS DE LA REDUCCIÓN

UNICIDAD DE ETIQUETAS

etiqueta ++

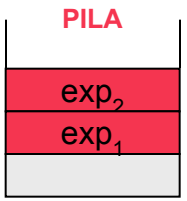
Generación de código para la comparación "<"

comparacion : exp_1 '<' exp_2

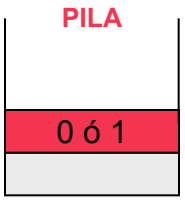
GENERACIÓN DE CÓDIGO

```
; cargar la segunda expresión en edx
pop dword edx
SI ($3.es_direccion == 1)
  mov dword edx , [edx]

; cargar la primera expresión en eax
pop dword eax
SI ($1.es_direccion == 1)
  mov dword eax , [eax]

; comparar y apilar el resultado
        cmp eax, edx
        jl near menor#
        push dword 0
        jmp near fin_menor#
menor#:     push dword 1
fin_menor#:
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN



DESPUÉS DE LA REDUCCIÓN

UNICIDAD DE ETIQUETAS

etiqueta ++

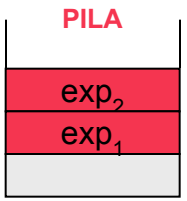
Generación de código para la comparación “>“

comparacion : exp_1 **‘>‘** exp_2

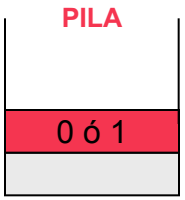
GENERACIÓN DE CÓDIGO

```
; cargar la segunda expresión en edx
pop dword edx
SI ($3.es_direccion == 1)
  mov dword edx , [edx]

; cargar la primera expresión en eax
pop dword eax
SI ($1.es_direccion == 1)
  mov dword eax , [eax]

; comparar y apilar el resultado
        cmp eax, edx
        jg near mayor#
        push dword 0
        jmp near fin_mayor#
mayor#:     push dword 1
fin_mayor#:
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN



DESPUÉS DE LA REDUCCIÓN

UNICIDAD DE ETIQUETAS

etiqueta ++

Introducción

- Hay una producción en la gramática para la operación de indexación de vectores:

  elemento_vector: TOK_IDENTIFICADOR '[' exp ']'

- El objetivo básico de la generación de código de la producción anterior es:
  - Generar código para **comprobar, en tiempo de ejecución, que el índice está dentro del rango permitido** según el tamaño del vector según se explicó en el apartado "Control de errores en tiempo de ejecución"
  - Generar código para dejar en la **cima de la pila la dirección del elemento indexado**.

Control de errores en tiempo de ejecución

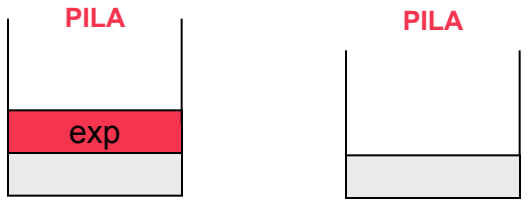
elemento_vector: **TOK_IDENTIFICADOR '[' exp ']'**

- El primer paso consiste en comprobar que el índice está dentro del límite permitido, es decir, entre 0 y el tamaño del vector menos 1 (ambos inclusive)

GENERACIÓN DE CÓDIGO

```
; Carga del valor del índice en eax
pop dword eax
SI ($3.es_direccion == 1)
  mov dword eax , [eax]

; Si el índice es menor que 0, error en tiempo de ejecución
cmp eax,0
jl near mensaje_1
; Si el índice es mayor de lo permitido , error en tiempo de
ejecución
cmp eax, <tamaño del vector –1>
jl near mensaje_1

; EN ESTE PUNTO EL ÍNDICE ES CORRECTO
; Y ESTÁ EN EL REGISTRO eax
...
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN

PUNTO INTERMEDIO EN LA REDUCCIÓN

Control de errores en tiempo de ejecución

- El código correspondiente a la etiqueta mensaje_1 sería:

```
mensaje_1:
        push dword mensaje_1
        call print_string
        add esp, 4
        jmp near fin
```

- La etiqueta fin está situada al final del programa, justo antes de la última instrucción ret.

Dirección del elemento indexado

- El segundo paso consiste en generar código para **dejar en la cima de la pila la dirección del elemento indexado**. Para ello hay que tener en cuenta las siguientes consideraciones:
  - En el segmento de datos *.bss* se reserva un bloque de memoria para almacenar todos los elementos del vector, y por lo tanto es necesario definir la disposición de dichos elementos dentro del bloque. Lo más razonable es ubicar los elementos por orden creciente de índice, es decir, primero el elemento 0, después el 1, y así sucesivamente hasta el elemento (tamaño-1).
  - Una vez que se ha elegido una disposición concreta, las operaciones de indexado deben de ser coherentes con la elección.
  - Cada elemento ocupa 4 bytes (porque así se ha hecho la reserva de memoria)
  - La dirección de inicio del vector es accesible mediante el nombre del vector.
  - Si se ubican los elementos por orden creciente de índice, la dirección del elemento **[i]** sería la siguiente:

    dirección elemento [i] = dirección_inicio_vector + i*4

Dirección del elemento indexado

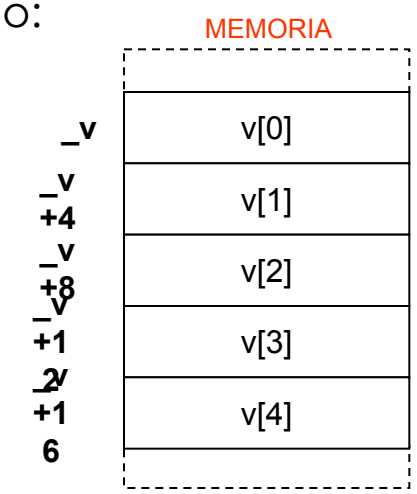
- Por ejemplo, la siguiente declaración de un vector de 5 posiciones en un programa ÓMICRON:

```
array int [5] v;
```

se traduce en la declaración ensamblador siguiente:

```
_v    resd 5
```

que reserva en memoria un bloque contiguo de 20 bytes, que se puede interpretar como 5 posiciones de 4 bytes cada una de ellas. Y si se considera que los elementos del vector se disponen dentro del bloque por orden creciente de índice, la memoria tendría el siguiente contenido:



Dirección del elemento indexado

GENERACIÓN DE CÓDIGO

OPCIÓN 1

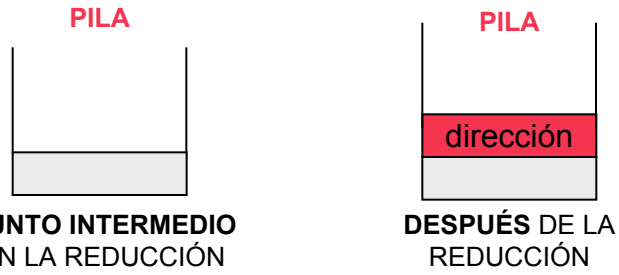
```
; Cargar en edx la dirección de inicio del vector
mov dword edx _$1.lexema
; Cargar en eax la dirección del elemento indexado
lea eax, [edx + eax*4]
; Apilar la dirección del elemento indexado
push dword eax
```

OPCIÓN 2

```
; Cargar 4 en edx (nº de bytes de cada elemento del vector)
mov dword edx, 4
; eax = eax*edx, es decir, eax = eax*4
imul edx
; Cargar en edx la dirección de inicio del vector
mov dword edx _$1.lexema
; Cargar en eax la dirección del elemento indexado
add eax, edx
; Apilar la dirección del elemento indexado
push dword eax
```

El registro `eax` contiene el valor del índice como consecuencia de la comprobación de la validez del valor del índice.

EVOLUCIÓN DE LA PILA



PUNTO INTERMEDIO EN LA REDUCCIÓN

DESPUÉS DE LA REDUCCIÓN

Introducción

- Las producciones de las sentencias de asignación son:
  - asignacion : TOK_IDENTIFICADOR '=' exp
  - asignacion : elemento_vector '=' exp

- Básicamente, la generación de código de las producciones anteriores tiene como objetivo escribir el **código ensamblador que hace efectiva la asignación**. Es decir, si por ejemplo la sentencia es una asignación de una constante a un identificador, la generación de código deberá producir las instrucciones en ensamblador que cargan el valor de la constante en la posición de memoria que ocupa el identificador.

- Las características de las partes izquierda y derecha de la asignación, determinan el conjunto de instrucciones en ensamblador que realizan de manera efectiva la asignación.

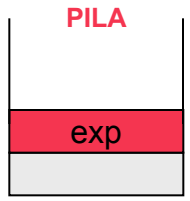
Asignación de una expresión a un identificador

asignacion: **TOK_IDENTIFICADOR '='** exp

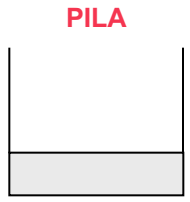
GENERACIÓN DE CÓDIGO

```
; Cargar en eax la parte derecha de la asignación
pop dword eax

SI ($3.es_direccion == 1)
   mov dword eax , [eax]

; Hacer la asignación efectiva
mov dword [_$1.lexema] , eax
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN



DESPUÉS DE LA REDUCCIÓN

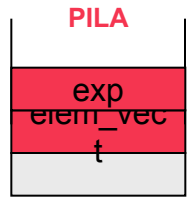
Asignación de una expresión al elemento de un vector

asignacion: elemento_vector **'='** exp

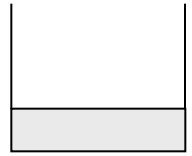
GENERACIÓN DE CÓDIGO

```
; Cargar en eax la parte derecha de la asignación
pop dword eax
SI ($3.es_direccion == 1)
    mov dword eax, [eax]

; Cargar en edx la parte izquierda de la asignación
pop dword edx

; Hacer la asignación efectiva
mov dword [edx] , eax
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN



DESPUÉS DE LA REDUCCIÓN

Introducción

- El lenguaje de programación ÓMICRON tiene una sola sentencia para la entrada de datos:

  lectura: TOK_SCANF TOK_IDENTIFICADOR

- En la acción semántica de la anterior producción, además de realizar las comprobaciones semánticas oportunas, se hace uso de la librería **olib.o** de la siguiente manera:
  - Si la lectura corresponde a un dato de tipo entero, se invoca a la función **scan_int**.
  - Si la lectura corresponde a un dato de tipo lógico, se invoca a la función **scan_boolean.**
  - Las funciones **scan_int** y **scan_boolean** tienen como parámetro de entrada la dirección de memoria que será destino del dato leído, por lo tanto, antes de invocar a una de ellas es necesario apilar la dirección adecuada.
  - Después de la invocación se restaura la pila (manipulando el puntero de pila o desapilando el parámetro apilado previamente)

Lectura de un identificador

lectura: **TOK_SCANF TOK_IDENTIFICADOR**

- En primer lugar, se prepara la pila para la llamada a la función correspondiente de la librería **olib.o**, apilando la dirección de memoria que será destino del dato leído:

```
push dword _$2.lexema
```

- El segundo paso consiste en invocar a la rutina de lectura adecuada al tipo del identificador:

```
call scan_int       ; si el identificador es de tipo entero
call scan_boolean   ; si el identificador es de tipo lógico
```

- Por último se restaura el puntero de pila:

```
add esp, 4
```

PENDIENTE 4: el identificador puede ser variable local, global o parámetro.

Introducción

- El lenguaje de programación ÓMICRON tiene una sentencia para escritura de expresiones. La producción correspondiente es:

escritura: TOK_PRINTF exp

Esquema de la acción semántica

- En la acción semántica de la producción correspondiente a la escritura de una expresión se hace uso de la librería **olib.o** de la siguiente manera:
  - Si la expresión es de tipo entero, se invoca a la función **print_int**.
  - Si la expresión es de tipo lógico, se invoca a la función **print_boolean.**
  - Las funciones **print_int** y **print_boolean** tienen un único parámetro que corresponde al **VALOR** de salida, por lo tanto, antes de invocar a una de ellas es necesario apilar dicho valor.
  - Después de la invocación se restaura la pila (manipulando el puntero de pila o desapilando el parámetro apilado previamente)
  - Una vez escrito el valor de salida, es conveniente realizar un salto de línea. La librería **olib.o** también proporciona la función **print_endofline** que escribe un salto de línea en la salida estándar. Esta función no tiene parámetros.

- Antes de reducir la producción, en la cima de la pila se encuentra el resultado de la reducción del no terminal *exp*.

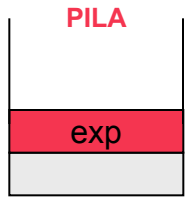
Esquema de la acción semántica

escritura: **TOK_PRINTF** exp

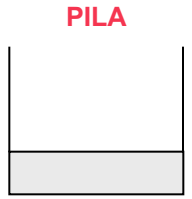
GENERACIÓN DE CÓDIGO

```
; Acceso al valor de exp si es distinto de constante
SI ($2.es_direccion == 1)
    pop dword eax
    mov dword eax , [eax]
    push dword eax

; Si la expresión es de tipo entero
call print_int
; Si la expresión es de tipo real
call print_float
; Si la expresión es de tipo lógico
call print_boolean

; Restauración del puntero de pila
add esp, 4

; Salto de línea
call print_endofnile
```

EVOLUCIÓN DE LA PILA



ANTES DE LA REDUCCIÓN



DESPUÉS DE LA REDUCCIÓN

Introducción

- El lenguaje de programación ÓMICRON tiene dos formatos de sentencia condicional descritos en las siguientes producciones:
  - condicional : TOK_IF '(' exp ')' '{' sentencias '}'
  - condicional : TOK_IF '(' exp ')' '{' sentencias '}' TOK_ELSE '{' sentencias '}'

- La generación de código de las sentencias condicionales tiene una característica particular que hasta ahora no se ha presentado en el desarrollo del compilador. La bifurcación del flujo de ejecución en función del valor de la condición requiere "recordar" las etiquetas de salto. En concreto, en la primera producción, una vez procesado el no terminal "exp", si su valor es VERDADERO, el flujo de ejecución continúa por las instrucciones correspondientes a la traducción del no terminal "sentencias", pero si su valor es FALSO, el flujo debe saltar a una etiqueta inmediatamente posterior a dicho bloque de instrucciones. El punto adecuado para escribir en ensamblador la instrucción de salto se encuentra antes de "sentencias", pero la escritura de la propia etiqueta a la que se salta sólo se puede hacer después de "sentencias". Por lo tanto, es necesario un mecanismo para asegurar que se usa el mismo nombre de etiqueta en los dos puntos en los que se hace uso de ella.

- Un posible mecanismo para "recordar" etiquetas consiste en:
  - **Modificar la gramática** de manera adecuada.
  - **Añadir un nuevo atributo** semántico que permita propagar etiquetas.

Condicionales simples

condicional : **TOK_IF** **'('** exp **')'** **'{'** ★ sentencias **'}'** ☆


```
;
;     exp
;
```

```
pop eax
mov eax, [eax]     ; si nivel de indirección
exp>0
cmp eax, 0
je near fin_si#
```

```
;
;     sentencias
;
```

```
fin_si#:
```

Estos dos bloques de código ensamblador se generan en momentos diferentes pero hacen referencia a una misma etiqueta (**fin_si#**)

Generación de código para sentencias condicionales

Condicionales simples

condicional : **TOK_IF** **'('** exp **')'** **'{'** ★ sentencias **'}'** ☆

MODIFICACIÓN DE LA GRAMÁTICA

if_exp: **TOK_IF** **'('** exp **')'** **'{'** ★

condicional: if_exp sentencias **'}'** ☆

★

Comprobaciones semánticas

Comprobar que $3.tipo == BOOLEAN

Reserva de etiqueta (#)

$$.etiqueta = etiqueta++

Generación de código

```
pop eax
mov eax , [eax]     ; Si exp es direccion
cmp eax, 0
je near fin_si#     ;   # es $$.etiqueta
```

☆

Generación de código

```
fin_si#:      ;   # es $1.etiqueta
```

Generación de código para sentencias condicionales

Condicionales compuestas

condicional: **TOK_IF** **'('** exp **')'** **'{'** ★ sentencias **'}'** ★ **TOK_ELSE** **'{'** sentencias **'}'** ☆


```
;
;     exp
;
```

```
pop eax
mov eax, [eax]     ; si nivel de indirección
exp>0
cmp eax, 0
je near fin_si#
```

```
;
;     sentencias
;
```

```
jmp near fin_sino#
fin_si#:
```

```
;
;     sentencias
;
```

```
fin_sino#:
```

Estos tres bloques de código ensamblador se generan en momentos diferentes pero hacen referencia a las mismas etiquetas (**fin_si#** y **fin_sino#**)

Generación de código para sentencias condicionales

Condicionales compuestas

condicional: **TOK_IF** **'('** exp **')'** **'{'** ★ sentencias **'}'** ★ **TOK_ELSE** **'{'** sentencias **'}'** ☆

MODIFICACIÓN DE LA GRAMÁTICA

if_exp: **TOK_IF** **'('** exp **')'** **'{'** ★

if_exp_sentencias : if_exp sentencias ★

condicional : if_exp_sentencias **TOK_ELSE** **'{'** sentencias **'}'** ☆

★

Comprobaciones semánticas

Comprobar que $3.tipo == BOOLEAN

Reserva de etiqueta (#)

$$.etiqueta = etiqueta++

Generación de código

```
pop eax
mov eax , [eax]     ;   Si exp es dirección
cmp eax, 0
je near fin_si#     ;   # es $$.etiqueta
```

★

Propagación de atributos

$$.etiqueta = $1.etiqueta

Generación de código

```
jmp near fin_sino#    ;   # es $1.etiqueta
fin_si#:              ;   # es $1.etiqueta
```

☆

Generación de código

```
fin_sino#:      ;   # es $1.etiqueta
```

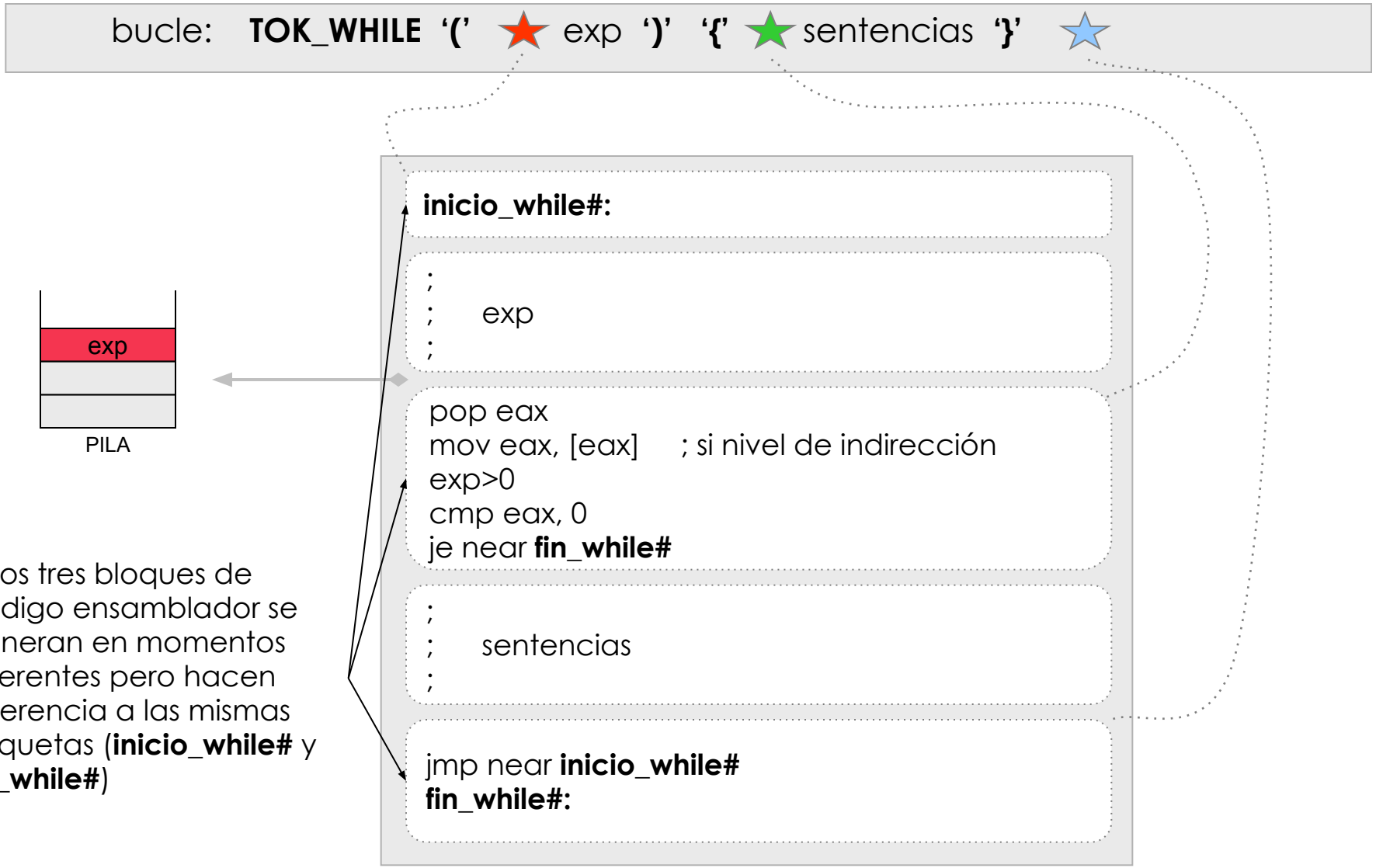
Introducción

- El lenguaje de programación ÓMICRON tiene una sentencia iterativa cuya sintaxis es la siguiente:

bucle: TOK_WHILE '(' exp ')' '{' sentencias '}'

- La traducción a ensamblador de las sentencias iterativas tiene características similares a la traducción de sentencias condicionales.
  La idea básica es la necesidad de implementar un mecanismo que permita "recordar" etiquetas.

Sentencia WHILE



Estos tres bloques de código ensamblador se generan en momentos diferentes pero hacen referencia a las mismas etiquetas (**inicio_while#** y **fin_while#**)

Sentencia WHILE

bucle: **TOK_WHILE** **'('** ★ exp **')'** **'{'** ★ sentencias **'}'** ☆

MODIFICACIÓN DE LA GRAMÁTICA

while : **TOK_WHILE** **'('** ★

while_exp : while exp **')'** **'{'** ★

bucle : while_exp sentencias **'}'** ☆

★ Comprobaciones semánticas

Comprobar que $2.tipo == BOOLEAN

Propagación de atributos

$$.etiqueta = $1.etiqueta

Generación de código

```
pop eax
mov eax , [eax]        ;   Si exp es dirección
cmp eax, 0
je near fin_while#     ;   # es $$.etiqueta
```

★ Reserva de etiqueta (#)

$$.etiqueta = etiqueta++

Generación de código

```
inicio_while#:         ;   # es $$.etiqueta
```

☆ Generación de código

```
jmp near inicio_while#     ;   # es $1.etiqueta
fin_while#:                ;   # es $1.etiqueta
```

Convenio de las llamadas

- Para que las llamadas a funciones trabajen de manera correcta, es necesario establecer lo que se conoce como **convenio de llamadas** que debe ser especificado explícitamente al diseñar el compilador.

- El convenio de llamadas tiene que dar respuesta a las siguientes preguntas:
  - ¿**Cómo se comunican los argumentos** desde el programa llamante a la función llamada?
  - ¿**Cómo se comunican los resultados** desde la función llamada hasta el programa llamante?

- Se recomienda utilizar el siguiente convenio de llamadas:
  - Para comunicar los argumentos desde el programa llamante a la función llamada:
    - Se utilizará la **pila**.
    - El programa llamante dejará en la pila el **valor** de los **argumentos** de la llamada **en el mismo orden** en el aparecen en la definición de la función.
    - **Tras la llamada**, el programa llamante tiene la responsabilidad de **eliminar los argumentos de la pila**.
  - Para comunicar los **resultados** desde la función llamada:
    - Se utilizará el registro **`eax`**.

Ejemplo

- Explicando más detalladamente el proceso consideremos el siguiente programa ÓMICRON.

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

Ejemplo

- Imaginemos el código NASM equivalente. Comencemos por el programa principal.

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

Ejemplo

- Las **secciones .bss y .data** junto con la **cabecera del segmento de código** tendrían el contenido que se muestra en la figura. Posteriormente estudiaremos la zona correspondiente al código de las funciones.

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
segment .bss
 _z resd 1
segment  .data
.....
segment .text
global main
extern ...
;
; código de las
; funciones
;
main:
```

Ejemplo

- La primera instrucción del programa principal es una sentencia de lectura que se traduce en NASM en una llamada a la función "scan_int". Antes de llamar a esta función, se apila su parámetro, que es la dirección donde debe depositar el dato.

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
...
main:
 push dword _z
```

El programa llamante introduce en la pila los argumentos de la función "scan_int"

Ejemplo

- Después de introducir en la pila el parámetro de la llamada, se invoca a la función.

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
...
main:
 push dword _z
 call scan_int
```

Realiza la llamada a la función

Ejemplo

- La función "scan_int" realiza su trabajo, y desde el principal se restaura la pila.

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
...
main:
 push dword _z
 call scan_int
 add esp, 4
```

El programa llamante restaura la pila manipulando el puntero de pila

Ejemplo

- La segunda sentencia del programa principal ÓMICRON es una instrucción de escritura, en la cual, antes de la propia escritura, se procesa la llamada a la función. Esta llamada implica realizar las mismas acciones que con la llamada a "scan_int"

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
...
main:
 push dword _z
 call scan_int
 add esp, 4



 push dword [_z]
 call _doble
 add esp, 4
```

De la misma manera hay qu
hacer con una llamada a un
función definida por el
programador de ÓMICRO

Ejemplo

- Siguiendo el convenio de llamadas previamente definido, **la función deja el retorno en el registro eax**. Por lo tanto, después de restaurar la pila, será necesario apilar en ella el contenido del registro eax (ya que la llamada a una función es una "exp")

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
...
main:
 push dword _z
 call scan_int
 add esp, 4



 push dword [_z]
 call _doble
 add esp, 4

 push dword eax
```

El retorno de la función se encuentra en `eax`

Ejemplo

- El código NASM correspondiente a la traducción de la sentencia de escritura “printf” y también el código del final del programa no son relevantes en la explicación actual y por ello se omiten en la figura.

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
...
main:
 push dword _z
 call scan_int
 add esp, 4




 push dword [_z]
 call _doble
 add esp, 4

 push dword eax

;
; resto de código
;
```

Ejemplo

- Para comprender el código que hay que generar para la función imaginemos **cómo está la pila** (donde se tiene que almacenar toda la información) en una llamada **si z contiene el valor 3**.

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

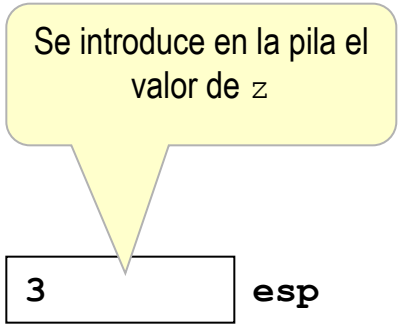
```
...
main:
 push dword _z
 call scan_int
 add esp, 4




 push dword [_z]
 call _doble
 add esp, 4

 push dword eax

;
; resto de código
;
```

Ejemplo

- Recuérdese que la pila de 32 bits tiene su **puntero** almacenado en el registro **esp** recuerde, también, que la pila "crece" hacia posiciones de memoria de valor menor, se representará gráficamente con crecimiento "hacia abajo".

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
...
main:
 push dword _z
 call scan_int
 add esp, 4



 push dword [_z]
 call _doble
 add esp, 4

 push dword eax

;
; resto de código
;
```

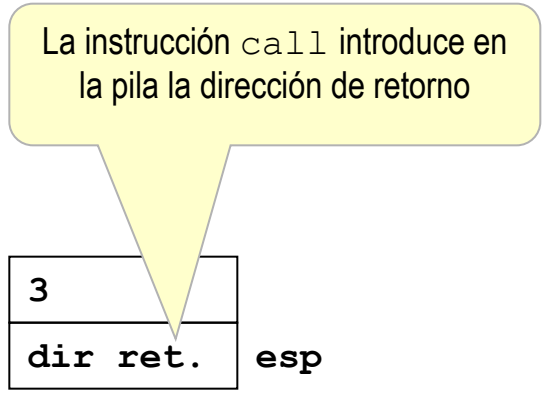


Ejemplo

- La instrucción **"call"** introduce en la **pila** la **dirección de retorno** a la que hay que regresar cuando termine la función y **salta** a la etiqueta indicada.

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
...
main:
 push dword _z
 call scan_int
 add esp, 4



 push dword [_z]
 call _doble
 add esp, 4

 push dword eax

;
; resto de código
;
```



Ejemplo

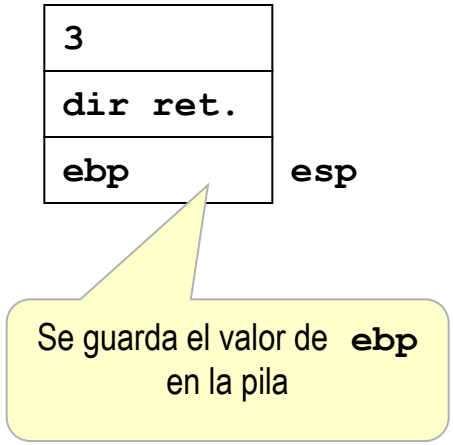
- Esta es la **pila** que se encuentra la **función** cuando **comienza su ejecución**.
- Está claro dónde se encuentra el argumento de la función (`arg`)
- Se tiene que utilizar también la **pila** para **guardar las variables locales** (`auxArg`)
- Debemos conocer qué expresión, en función de los **registros de gestión de la pila**, sirve para localizar en ella **argumentos** y **variables locales**.
- Obsérvese que el compilador tendrá que generar el código necesario para el cuerpo de la función.
- Por las decisiones de diseño tomadas, se utilizará la **pila como almacenamiento temporal**, por lo tanto, el valor del registro **`esp`** **cambiará**.
- NASM proporciona otro registro, en concreto, el registro **`ebp`**, específicamente para poder solucionar esta circunstancia.
- Para ello, hay que hacer lo siguiente:
  - **Guardar** el valor de **`ebp`** (también en la pila)
  - Utilizar **`ebp`** para mantener una **copia del valor de `esp`** en el momento de la entrada en la función
  - Utilizar **`ebp`** para **localizar** los **parámetros** y las **variables locales**
  - **Recuperar** los valores de **los dos registros** antes de salir de la función

Ejemplo

- La **primera acción** en la generación de código del cuerpo de una función es **salvar en la pila el registro ebp**, ya que se va a utilizar como puntero base para localizar en la pila los argumentos y las variables locales de la función.

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
_doble:
 push ebp
```

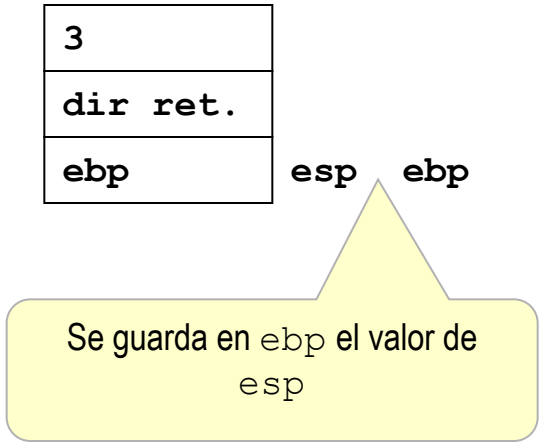


Ejemplo

- La **segunda acción** es **copiar en el registro ebp el valor del puntero de pila**, para poder gestionar el acceso a los parámetros y las variables locales (se verá posteriormente)

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
_doble:
 push ebp
 mov ebp, esp
```

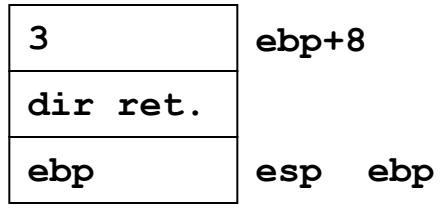


Ejemplo

- Recuérdese que usamos un tamaño de dato de 32 bits, que son 4 bytes. Así es fácil saber que `arg` se encuentra separado de ebp por 2 dword, es decir + 8 bytes

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
_doble:
 push ebp
 mov ebp, esp
```

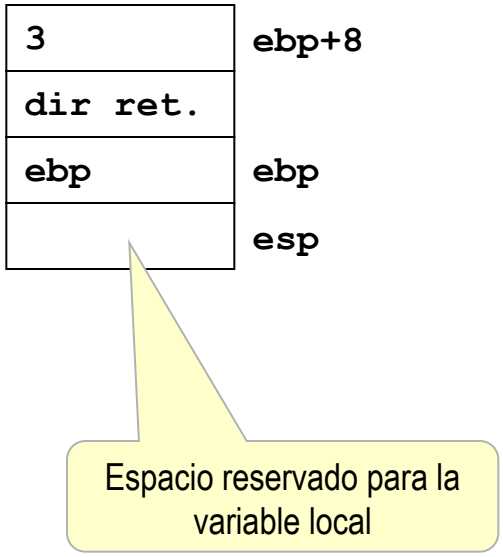


Ejemplo

- Ahora sólo queda localizar (**reservar espacio en la pila** y saber como acceder posteriormente) las **variables locales** (`auxArg`). Para ello se utiliza las posiciones libres bajo la cima de la pila (restamos a `esp` lo necesario)

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
_doble:
 push ebp
 mov ebp, esp
 sub esp, 4
```



Ejemplo

- Por lo tanto, la variable local `auxArg` se encontrará siempre 4 bytes por debajo de `ebp` es decir, en `ebp-4`.

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
_doble:
 push ebp
 mov ebp, esp
 sub esp, 4
```

Ejemplo

- Ahora se puede generar el código para el cuerpo de la función (aquí se genera "a mano" y es diferente de lo generado automáticamente por el compilador)

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
_doble:
 push ebp
 mov ebp, esp
 sub esp, 4


 mov dword eax, [ebp+8]
 mov dword [ebp-4], eax
 mov dword edx, 2
 imul edx
```

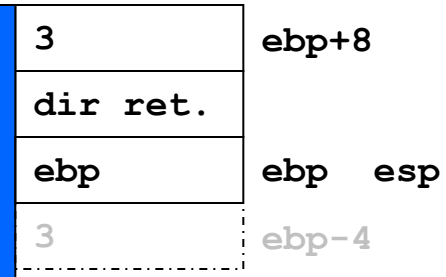
La operación `imul` multiplica el registro `eax` por el registro que se pasa a la operación, en este caso `edx` y deja el resultado en `eax`

Ejemplo

- Antes de salir de la función se debe **"recuperar"** los valores iniciales de **ebp** y **esp**. Recuérdese que en `ebp` se guardó el valor de `esp`, y por lo tanto recuperar el valor de `esp` consiste en un simple movimiento (`mov esp, ebp`)

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
_doble:
 push ebp
 mov ebp, esp
 sub esp, 4


 mov dword eax, [ebp+8]
 mov dword [ebp-4], eax
 mov dword edx, 2
 imul edx

 mov esp, ebp
```



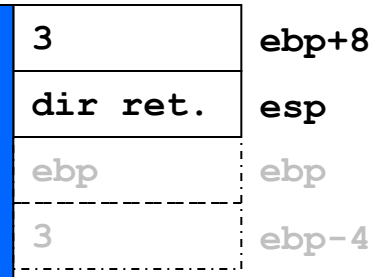
Se copia en `esp` su valor "inicial", que está guardado en `ebp`. Obsérvese que es como si se hicieran los `pop` necesarios para eliminar las variables locales

Ejemplo

- Para **recuperar** el valor que tenía el registro **ebp** al inicio de la función, simplemente se accede a la cima de la pila, ya que se apiló previamente.

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
_doble:
 push ebp
 mov ebp, esp
 sub esp, 4


 mov dword eax, [ebp+8]
 mov dword [ebp-4], eax
 mov dword edx, 2
 imul edx

 mov esp, ebp
 pop ebp
```



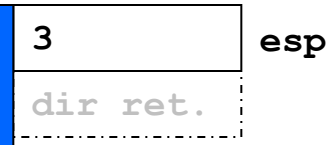
Se recupera (desde la pila) el antiguo valor de `ebp`. Obsérvese que, tras modificar en la instrucción anterior `esp` la cima de la pila apunta ahora a la posición que guarda la copia de `ebp`

Ejemplo

- Como **última instrucción de la función**, la instrucción **"ret"**, que coge de la pila la dirección de retorno que apiló la instrucción "call"

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
_doble:
 push ebp
 mov ebp, esp
 sub esp, 4



 mov dword eax, [ebp+8]
 mov dword [ebp-4], eax
 mov dword edx, 2
 mul edx

 mov esp, ebp
 pop ebp
 ret
```



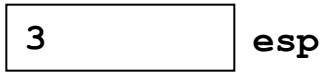
Y ya se puede devolver el control al programa llamante mediante la instrucción `ret`. Esta instrucción elimina de la pila la dirección de retorno

Ejemplo

- Retornamos, por lo tanto, al código del programa principal que se encuentra la pila como la tenía antes de la ejecución de la instrucción `call`. Obsérvese que la pila todavía contiene los argumentos de la función, el programa debe **restaurar la pila**.

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
...
main:
 push dword _z
 call scan_int
 add esp, 4




 push dword [_z]
 call _doble
 add esp, 4

 push dword eax

;
; resto de código
;
```

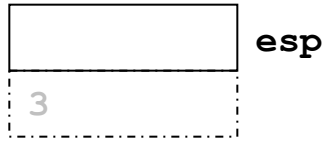


Ejemplo

- Para restaurar la pila se manipula su puntero, y se posicionará en el lugar donde estaba antes de producirse la inserción de los parámetros de la llamada a la función.

```
main {
 int z;

 function int doble(int arg)
 {
   int auxArg;

   auxArg = arg;
   return 2*arg ;
 }

 scanf z;
 printf doble(z)
}
```

```
...
main:
 push dword _z
 call scan_int
 add esp, 4



 push dword [_z]
 call _doble
 add esp, 4

 push dword eax

;
; resto de código
;
```



Generación de código para las llamadas a funciones

- Lo observado en el ejemplo anterior puede generalizarse para cualquier función con cualquier número de variables locales y cualquier número de argumentos.

- El programa llamante debe primero **copiar los argumentos de la función en la pila en el mismo orden en el que aparecen en la declaración de la función**

```
push dword <1er argumento>
    …
push dword <último argumento>
```

- Las producciones relacionadas con los argumentos de llamada a una función son las siguientes:

exp: TOK_IDENTIFICADOR '(' lista_expresiones ')'
lista_expresiones: exp resto_lista_expresiones
lista_expresiones:
resto_lista_expresiones: ',' exp resto_lista_expresiones
resto_lista_expresiones:

Generación de código para las llamadas a funciones

- Como puede observarse en las producciones anteriores, **los parámetros de llamada a una función son una lista de no terminales "exp".**

- Por lo tanto, para cumplir el **convenio de llamadas** en el que se dice que **los parámetros se pasan por valor**, es necesario controlar que las reducciones del no terminal "exp" que ocurran dentro de una lista de parámetros de llamada dejen en la pila un valor y nunca una dirección.

- Por ejemplo, la producción siguiente:

  exp: TOK_IDENTIFICADOR

  si se reduce dentro de una lista de parámetros de llamada, **debe dejar en la pila el valor del identificador y no su dirección.**

- Para implementar esta característica del convenio de llamadas, basta con diseñar un mecanismo que permita saber en las reducciones del no terminal "exp" si está ocurriendo dicha reducción dentro de una lista de parámetros de llamada a función o no, y en caso afirmativo dejar en la pila el valor y no la dirección.

Generación de código para las llamadas a funciones

- De todas las reglas del no terminal "exp", solamente hay dos que dejen en la pila una dirección en lugar de un valor, y son las siguientes:

  exp: TOK_IDENTIFICADOR
  exp: elemento_vector

- Por lo tanto, sólo en estas producciones hay que controlar si la reducción ocurre dentro de una lista de parámetros de llamada a función.

- Las demás producciones del no terminal "exp" siempre dejan un valor en la pila y por lo tanto su reducción siempre trabaja de la misma manera. Estas reglas son:

  exp: exp '+' exp (y todas las demás reglas de operaciones aritmético-lógicas)
  exp: constante
  exp: '(' comparacion ')'
  exp: TOK_IDENTIFICADOR '(' lista_expresiones ')'

Generación de código para las llamadas a funciones

- El programa llamante, después de depositar en la pila los parámetros de la llamada a la función debe **llamar a la función**:

```
call _<nombre función>
```

- Posteriormente **restaurar la pila**.

```
add esp, <4 * nº argumentos de la función>
```

- Como la función, atendiendo al convenio de llamadas, deja el resultado en eax, y siempre la llamada a una función aparece en el lugar de una expresión, es necesario, después de que la función termine, apilar el la pila el contenido de eax.

```
push dword eax
```

- La producción adecuada para realizar estas tres acciones se muestra en la siguiente figura, en la que también se incluyen acciones relacionadas con el análisis semántico.

Generación de código para las llamadas a funciones

exp: idf_llamada **'('** lista_expresiones **')'**

En la reducción de este nuevo símbolo se comprueba, entre otras cosas, que el identificador existe y que pertenece a una función.

¿número de parámetros formales = actuales?

SI

NO

Mostrar mensaje de error semántico y terminar con error

Generación de código

```
call _$1.lexema
add esp, 4* global_num_parametros_llamada_actual
push dword eax
```

Cálculo de atributos

```
$$.tipo = tipo del elemento buscado en la tabla
$$.es_direccion = 0
```

Actualización de variables globales

```
en_explist = 0
```

La variable "en_explist" es la variable global que permite controlar si el proceso de compilación se encuentra dentro de una lista de parámetros de una llamada a función

Código inicial y final en el cuerpo de una función

- Las **primeras sentencias NASM del cuerpo de cualquier función** deben ser las siguientes:

```
_<nombre_funcion>:
    push ebp
    mov ebp, esp
    sub esp, <4*nº variables locales>
```

- El punto adecuado para generar las primeras líneas de código de la función sería el indicado con el símbolo ❷ en la producción. Dicho punto también se identificó en el estudio del análisis semántico como lugar adecuado para la realización de acciones. Por lo tanto, la gramática ya está modificada.

(22) funcion : TOK_FUNCTION tipo TOK_IDENTIFICADOR
'(' parametros_funcion ')' '{' declaraciones_funcion ❷
sentencias '}'

Código inicial y final en el cuerpo de una función

- **Las últimas** instrucciones en el cuerpo de una función deben ser las siguientes:

```
mov esp, ebp
pop ebp
ret
```

- La producción adecuada para escribir estas últimas instrucciones del cuerpo de una función se muestra en la siguiente figura, en la que también se incluyen acciones relacionadas con el análisis semántico.

Código inicial y final en el cuerpo de una función

retorno_funcion : **TOK_RETURN** exp

Se accede a la información de la función en la tabla de símbolos para comprobar que "exp" tiene el mismo **tipo** que el retorno de la función.

Comprobación de tipo y direcciones

Las **direcciones** de exp siempre tienen que ser 0 ó 1.

¿OK ?

SI

NO

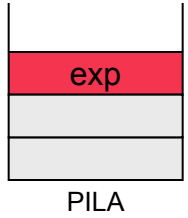
Actualización de variables globales

fn_return ++

Mostrar mensaje de error y terminar con error

Generación de código

```
pop dword eax
SI ($2.es_direccion==1)
   mov eax , [eax]
mov dword esp, ebp
pop dword ebp
ret
```



La variable **fn_return** permite controlar que siempre haya al menos una instrucción de retorno en el cuerpo de una función

Localización del contenido de los parámetros

- En **el código del cuerpo de la función**, cuando se quiera acceder al valor de los parámetros se utilizará la expresión

`[ebp +` <4 + 4*(numero de parámetros -posición del parámetro en declaración)>`]`

- Esta expresión de acceso es válida siempre que el **primer parámetro** tenga asignada la **posición 0**.

- Recuerde que, el primer 4 es para "saltar" la dirección de retorno.

- Por ejemplo:
  - El valor del último argumento será `[ebp+8]`.
  - El valor del penúltimo argumento será `[ebp+12]`.
  - …
  - El valor del i-esimo argumento será `[ebp+`<4+4*(numero de parámetros – i)>`]`.

Localización del contenido de las variables locales

- En **el código del cuerpo de la función**, cuando se quiera acceder al valor de las variables locales se utilizará la expresión

`[ebp – <4*posición de la variable en declaración>]`

- Esta expresión de acceso es válida siempre que la **primera variable local** tenga asignada la **posición 1**.

- Por ejemplo:
  - El valor de la primera variable local será `[ebp-4]`.
  - El valor de la segunda variable local será `[ebp-8]`.
  - …
  - El valor de la i-esima variable local será `[ebp-<4*i>]`.

Localización de la dirección de parámetros y variables globales

- Sin embargo, como se ha indicado en la generación de código de otras construcciones de ÓMICRON (por ejemplo en el de las expresiones que contienen identificadores), a veces es necesario utilizar "la dirección" y no "el contenido"

- Recuérdese, por ejemplo, que para la variable global ÓMICRON `z`,
  - La expresión para referirse al contenido es

```
[_z]
```

  - Pero, la expresión para la dirección es

```
_z
```

- Las funciones no utilizan nombres explícitos para los parámetros y las variables locales, por lo que la expresión equivalente para su dirección (`ebp+`<desplazamiento> y `ebp-`<desplazamiento>, respectivamente) son incorrectas en NASM.

- NASM proporciona la instrucción `lea` para gestionar esta cuestión.

Localización de la dirección de parámetros y variables globales

- En este curso se va a utilizar la siguiente sintaxis para la instrucción `lea` de NASM

```
lea <registro> [<expresión>]
```

- Donde
  - <registro>, es el nombre de un registro (por ejemplo `eax`)
  - <expresión>, puede ser, por ejemplo `ebp+12`.

Tratamiento de identificadores

- Como resumen de estos últimos puntos, considérese el identificador ÓMICRON `z`. Se utilizará las siguientes expresiones NASM para
  - Acceder a su contenido
    - Si es una variable global

```
[_z]
```

    - Si es el argumento i-ésimo de una función (el primer argumento es el número 0)

```
[ebp+<4+4*(total argumentos – i)>]
```

    - Si es la variable local i-ésima de una función (la primera variable local es la número 1)

```
[ebp-<4*i>]
```

Tratamiento de identificadores

- Acceder a su dirección
  - Si es una variable global

```
_z
```

  - Si es el argumento i-ésimo de una función, lo siguiente deja en el registro `eax` la dirección

```
lea eax, [ebp+<4+4*(número parámetros –i)>]
```

  - Si es la variable local i-ésima de una función

```
lea eax, [ebp-<4*i>]
```

Tratamiento de identificadores

- Es **muy importante que el alumno propague este tratamiento a todos los lugares donde pueda aparecer un identificador**, exceptuando:
  - Identificadores de vectores porque nunca pueden ser variables locales o parámetros.
  - Identificadores de funciones.

- Por lo tanto, la propagación del tratamiento se realizará en las siguientes producciones:

asignacion: TOK_IDENTIFICADOR '=' exp
lectura: TOK_SCANF TOK_IDENTIFICADOR
exp: TOK_IDENTIFICADOR

Tratamiento de identificadores

- **En resumen**, en las producciones anteriormente indicadas, se tiene que determinar para el identificador que aparece:
  - Si es
    - una variable global
    - una variable local de una función
    - un argumento de una función
  - Si el código que se necesita generar accederá
    - A su contenido
    - A su dirección

Ya que cada opción necesita un tratamiento diferente.