Universidad Complutense de Madrid

Procesadores del Lenguaje

Doble Grado en Ingeniería Informática - Matemáticas

Práctica Final

Autores:
Javier Guzmán Muñoz
Jorge Villarrubia Elvira

13 de junio de 2020

1. Introducción

Hemos desarrollado un lenguaje imperativo similar a lenguajes del tipo C/C++. Algunas de sus características principales son:

- Los programas se escriben en **un único fichero fuente**.
- **No es necesario indicar un punto de entrada** (un main) a los programas. Las instrucciones se ejecutan en el orden en que aparecen en el fichero de código, exceptuando las instrucciones que pertenezcan a cuerpos de funciones o procedimientos que se ejecutarán sólo cuando se llame a los mismos.
- Así, entendemos que la función main es el fichero código en sí y sobre la que se declaran el resto de bloques que a continuación detallaremos.
- Los **comentarios** permitidos en los ficheros fuente son sólo de línea y comienzan con el símbolo `#`.
- Todos aquellos aspectos para los que no se indica alguna restricción explícita en este documento **se consideran permitidos** en nuestro lenguaje
- Aunque ya es sabido recordamos que sólo es posible generar código para los **tipos primitivos enteros y booleanos** pues la máquina-P no soporta otros.
- Se suministran unos **juegos de prueba** que permiten probar todo lo aquí descrito y ver todas las posibilidades que nos ofrece el lenguaje.

2. Tipos

Nuestro lenguaje cuenta con los tipos usuales de los lenguajes de programación:

- **Enteros:** se identifican con la palabra reservada `int`.
- **Reales:** Se separa la parte decimal de la entera con un punto (.). Se identifican con la palabra reservada `float`.
- **Booleanos:** Se incluyen los valores `true` y `false` como palabras reservadas, además del identificador de tipo `bool`.
- **Caracteres:** Se representan mediante con el símbolo del carácter entre comillas simples (por ejemplo, 'a'). Se identifican mediante la palabra reservada `char`.
- **Vectores n-dimensionales:** Se identifican mediante la palabra reservada `vector`. Veamos algunos detalles de este tipo:

- Para declararlos hay que indicar entre `<,>` el tipo del que estamos creando el vector.
- Un vector **puede ser de cualquiera de los tipos que soporta el lenguaje**, incluyendo los no primitivos y evidentemente podemos tener vectores de vectores creando así matrices con el número de dimensiones que se quiera.
- Así, declararemos un objeto de tipo vector indicando el tipo del mismo como `vector<tipo_vect>` donde `tipo_vect` es cualquier tipo permitido.
- Siempre que declaremos vectores debemos **indicar el tamaño y el valor inicial** de los mismos. Para ello se proporciona la función `creaVector(valorIni, tam)` que igualaremos a la declaración de una variable de tipo vector.
- Para el caso de vectores de más de una dimensión el tipo inicial tiene que ser un `creaVector` en el que se indique el valor inicial y el el tamaño en esta dimensión. Así, en la parte de `valorIni` anidaremos tantas veces la función `creaVector` hasta llegar a un tipo que no sea un vector, para el cual indicamos una expresión cualquiera de ese tipo que se asignará como valor inicial a todas las posiciones del vector o matriz.
- Por ejemplo podemos declarar una matriz de de enteros, con dimensiones 3*4 y valor inicial 0 como:

```
vector<vector<int>> v = creaVector(creaVector(0,4),3);
```

- No se puede dar valor inicial a un vector asignándole directamente otro vector aunque sea de su mismo tipo, la única declaración permitida (y además obligatoria) es la expuesta antes.
- Se permiten **vectores dinámicos**, entendiéndose como tales aquellos para los que no indicamos explícitamente su tamaño como un entero. Estos vectores se guardarán en la parte del heap de la máquina-P reservando el tamaño correspondiente en tiempo de ejecución
- Estos vectores y matrices dinámicas **se crean del mismo modo que los estáticos**, concatenando tantas veces la función `creaVector` como queramos y ahora pudiendo indicar cualquier expresión entera como tamaño para cada una de las dimensiones. Por ejemplo la siguiente es una declaración correcta de un vector dinámico:

```
int x = 3;
int y = x;
vector<vector<int>> v = creaVector(creaVector(0,x+y),3*y);
```

- **Struct o registros:** Se permiten definiciones por el usuario de tipos estructurados con campos con nombre propio que pueden ser de cualquiera de los demás tipos permitidos, incluidos otros registros. Para definirlos se

emplea la palabra reservada `struct` seguida del nombre que le queremos dar al tipo y entre llaves incluimos la declaración desus campos indicando tipo y nombre. Adicionalmente **podemos dar valor inicial por defecto a los campos** que queramos, **siendo obligatorio** el correspondiente `creaVector` **para los campos que sean de tipo vector.** Un ejemplo de declaración de un tipo estructurado sería:

```
struct persona{
    int dni;
    int edad = 20;
    vector<int> codigos = creaVector(0,5);
}
```

- **Punteros:** Se permiten **apuntadores a cualquiera de los tipos definidos** previamente, incluidos apuntadores a apuntadores. Algunas características de este tipo son:
  - Para declararlos se debe indicar el tipo al que apuntan de la siguiente manera, con el nombre del tipo apuntado entre menores: `<tipo_apuntado>* nombre_puntero`.
  - Podemos asignar un apuntador a otro (siempre que sean del mismo tipo, tanto en declaraciones como asignaciones).
  - Se proporciona **una función predefinida** para reservar memoria dinámica.
  - Esta función se le asigna a un variable de un puntero **en su declaración** (**no puede usarse fuera de aquí**) y su sintaxis es la palabra reservada `new` seguida del tipo apuntado y unos paréntesis.
  - Si el tipo apuntado es un vector o matriz entre estos paréntesis se deberá indicar **un entero por cada una de las dimensiones del vector o matriz**. Sólo se permiten enteros aquí e indican el valor de cada una de las dimensiones de la matriz. No se permitirá ningún otro valor y del mismo modo no se permitirán valores aquí en news que no sean a punteros a vectores.
  - Estos son algunos ejemplos de declaraciones válidas de punteros:

```
<int>* p = new int();
<int>* q = p;
vector<vector<int>>* pv = new vector<vector<int>>(5,3);
```

- **Alias de tipos:** Se permiten **redefinir nuevos tipos** a partir de tipos básicos dándoles un nuevo nombre que luego podemos usar en declaraciones como si de un identificador de tipo se tratase.
  - A efectos de compilación **no son en verdad nuevos tipos**, sino alias que facilitan la labor al programador, pero desde el punto de vista de la compilación son indistinguibles del tipo que redefinen.

- De hecho, cuando nos encontramos declaraciones de variables de este tipo recorriendo el árbol sintáctico del programa los sustituimos por el tipo primitivo que representan.
- Para definir estos alias usamos la palabra `typedef` seguida del tipo primitivo y el nuevo nombre que le vamos a dar. Por ejemplo:

```
typedef int dni;
typedef dni identificacion;
typedef vector<vector<int>> matriz;
typedef <int>* punt_a_int;
```

- **Tipos enumerados:** Se define un conjunto de identificadores que serán los valores de este tipo. Algunos aspectos relevantes de este tipo son:
  - Internamente **son tratados como enteros**, asignándole a cada valor del enumerado un entero **comenzando desde 0** y en el orden en que aparecen en su declaración.
  - En tiempo de compilación **son indistinguibles de los enteros**, pudiéndose usar como tales y aparecer en cualquier expresión o parte de una expresión en la que pueda aparecer un entero.
  - De hecho las variables declaradas con este tipo o los valores del mismo no son tratados como internamente como un tipo distinto sino como enteros.
  - En tiempo de compilación se sustituyen todas las variables de este tipo por enteros y todos los valores por el correspondientes enteros según el orden en el que fueron declarados.
  - Para declarar un tipo enumerado se usa la palabra reservada `enum` seguida del nombre del tipo y sus valores entre llaves y separados por barras (`|`).
  - Para usar los valores de este tipo se usa el nombre del tipo, un punto y el nombre del valor.
  - El siguiente ejemplo muestra el uso de los enumerados:

```
enum diaSemana = {L|M|X|J|V|S|D};
diaSemana martes = disSemana.M;
diaSemana jueves = martes + 2;
```

3. Instrucciones

El lenguaje cuenta con las siguientes instrucciones, que son básicamente aquellas con las que cuenta cualquier lenguaje de programación del estilo de C:

- **Instrucción de declaración:** declaramos una variable con un identificador y un tipo indicados en la declaración. Admite dos modalidades, declaraciones sin valor inicial y declaraciones con valor inicial:

```
# Declaracion sin valor inicial
int x;

# Declaracion con valor inicial
bool y = 3 > 4;
```

Esta instrucción presenta unas restricciones mínimas que detallamos a continuación:

- Como ya se indicó en la sección, **las declaraciones de vectores deben llevar parte de valor inicial**, que consistirá en tantas funciones `creaVector` como dimensiones tenga el vector declarado, independientemente de si indicamos tamaños estáticos o dinámicos. No está permitida otra declaración para los vectores y matrices. Así, el esquema de declaración de un vector es el siguiente:

  ```
  vector<...<vector<tipo_vector>>...> =
     creaVector(...(creaVector(valor_ini, dim_n)...),dim_1);
  ```

  donde `tipo_vector` es cualquiera de los demás tipos permitidos por el lenguaje.
- En las declaraciones de punteros **se puede reservar memoria dinámica** para ese puntero con la instrucción `new` que se explicó en el apartado anterior. Esta instrucción sería el valor inicial que le asignaríamos al puntero en su declaración, pero **no es obligatoria**, podemos declarar punteros sin valor inicial o igualarles en su declaración a otro puntero de su mismo tipo pasando entonces las dos variables a apuntar al mismo objeto de memoria dinámica. Por otro lado, **no se permite usar la instrucción `new` fuera de las declaraciones**.
- Podemos dar valor inicial a las variables de un tipo estructurado **asignándole en la declaración otro objeto de su mismo tipo** y lo que haremos será crear **una copia** en memoria de los campos de ese otro registro que le estamos asignando. Convienen recalcar que **las declaraciones será el único lugar en el que se permitan estas copias.**

- **Instrucción de asignación:** Estas instrucciones permiten asignar valores a variables de tipos simples, posiciones concretas de vectores o matrices y a campos de estructuras. Constan de una parte izquierda y una parte derecha:

  - La **parte izquierda** de las asignaciones puede ser un identificador de una variable con tantos puntos (para acceder a campos de estructuras) y corchetes (para acceder a posiciones de vectores) como se quiera. También podemos acceder aquí al valor apuntado por un puntero, para ello proporcionamos el operador asterisco (*) que se coloca delante del identificador de la variable de tipo puntero.

- La **parte derecha** es una expresión cualquiera cuyo tipo final sea el mismo que el de la variable (opcionalmente con puntos y corchetes o acceso a puntero) de la izquierda.
- La asignación puede ser de cualquiera de los tipos permitidos en nuestro lenguaje **exceptuando los vectores y los tipos estructurados (registros)**, que deberemos asignarles valores o bien en su declaración o bien campo a campo.
- La instrucción `new` de creación de memoria dinámica no se permite en asignaciones, **solo en declaraciones**.
- **Se puede asignar una variable de tipo puntero** a otra del mismo tipo, pasando así ambas a apuntar al mismo objeto en memoria dinámica, si es que el puntero de la parte derecha apuntaba a algo, si no ambos apuntarán a `null`.
- El siguiente ejemplo muestra una asignación válida, suponiendo que los tipos de los identificadores son los correctos:

```
personas[0].datos.nombres[1][0].dni = 3*5+x-3%2;
```

- **Instrucciones de bucle:** Se incluyen los dos bucles habituales en los lenguajes de programación:
  - **Bucle while:** Se ejecuta su cuerpo mientras se verifique una condición booleana. Su cuerpo son varias instrucciones que se ejecutan después de comprobar una condición en cada iteración:

```
while (cond_booleana) {
    cuerpo_del_bucle
    ...
}
```

  - **Bucle for:** Sigue la siguiente estructura:

```
for (dec_o_asig_ini; cond_booleana; paso){
    cuerpo_del_bucle
    ...
}
```

    donde `dec_o_asig_ini` es una instrucción de declaración de una variable con valor inicial o una asignación a una variable previamente declarada, `cond_booleana` es la condición que se comprueba en cada iteración para decidir la terminación del bucle y `paso` es una asignación que se ejecuta en cada vuelta del bucle.

- **Instrucción condicional:** Sigue el esquema típico de las instrucciones `if-else`. Así cuenta con la parte del if que siempre aparece y consta de una serie de instrucciones (cualesquiera) que se ejecutan si se verifica una condición booleana. **La parte del else es opcional** (no tiene por qué incluirse siempre y en caso de aparecer las instrucciones de su cuerpo se

ejecutan si no se verifica la condición booleana. Así tenemos dos esquemas para esta instrucción:

- Instrucción `if`:

```
if (con_booleana){
    cuerpo_if
    ...
}
```

- Instrucción `if-else`:

```
if (con_booleana){
    cuerpo_if
    ...
}
else {
    cuerpo_else
    ...
}
```

- **Instrucción de Switch:** su funcionamiento es idéntico al de esta instrucción en cualquier otro lenguaje de programación. Así, tenemos una expresión que se evalúa y varios casos con expresiones del mismo tipo que la expresión del switch. Se va comparando el valor de la expresión principal con la de cada caso y se ejecutan las instrucciones del cuerpo de ese caso. Su esquema general es el siguiente:

```
switch (exp_switch){
    case exp_1:
        cuerpo_case_1
        ...
        break;
     case exp_2:
        cuerpo_case_2
        ...
        break;

    ...

    case exp_n:
        cuerpo_case_n
        ...
        break;
    default:
        cuerpo_default
        ...
}
```

Hacemos a continuación algunas aclaraciones importantes sobre esta instrucción:

- Como se puede apreciar en el esquema anterior **obligatoriamente** todos los case acaban con la instrucción `break;`.

- El caso por defecto (`default:`) **es siempre obligatorio**, pudiéndose dejar vacío si no queremos considerarlo.
- La generación de código de eficiencia optimizada que se nos proponía para esta parte era demasiado restrictiva (pues se nos limitaba a casos con enteros consecutivos) y decidimos **no imponer estas restricciones a nuestro lenguaje** dando más libertad al programador.
- Así, a la hora de generar código **se traduce directamente la instrucción a una serie de if anidados** con cada caso en los que las condiciones son comprobaciones de igualdad del valor de la expresión del switch y la que define cada case.
- En este sentido somos más flexibles que otros lenguajes de programación y permitimos que las expresiones en los cases no sean un valor concreto sino que **pueden ser cualquier expresión evaluable.**
- Podemos hacer switch de expresiones de cualquier tipo básico, incluidos los enumerados (que a todos los efectos se comportaban como enteros).

- **Instrucción para la declaración de tipos estructurados:** Se declara un tipo estructurado y sus campos. Las declaraciones de los campos son instrucciones de declaración con las mismas restricciones que estas instrucciones tenían. Estas declaraciones pueden ser de cualesquiera de los tipos definidos en ese momento. Esta instrucción **no acaba en punto y coma**, pero sí lo hacen las declaraciones de sus campos. Su esquema es el siguiente:

```
struct nombre_registro{
     declaracion_campo_1
     ...
     declaracion_camapo_n
 }
```

- **Instrucción de declaración de alias de tipos:** Se puede declarar un alias de cualquiera de los tipos simples y compuestos, incluyendo otros alias o registros. Su sintaxis es:

```
typedef tipo_antiguo nuevo_nombre;
```

- **Instrucción de declaración de tipos enumerados:** Se pueden declarar tipos enumerados que a todos los efectos tendrán las mismas propiedades con los enteros y se podrán usar indistintamente unos u otros en las expresiones de los dos tipos. Su sintaxis es la siguiente:

```
enum nombre_enum = {nombre_valor_1 | ... | nombre_valor_n};
```

- **Instrucción de declaración de funciones:** Podemos declarar funciones cuyo código se generará pero no se ejecutará hasta que no se la llama como parte de una expresión. Su sintaxis es la siguiente:

```
fun tipo_ret nombre_func (tipo_1 param_1,...,tipo_n param_n){
    ...
    cuerpo_funcion
    ...
    return exp_retorno;
}
```

Hacemos a continuación algunas aclaraciones sobre esta sintaxis:

- Por las propias limitaciones de la máquina-P proporcionada el tipo de retorno de las funciones **sólo puede ser un tipo básico** (ya sea el tipo en sí o un alias de uno de estos tipos declarado previamente vía `typedef`.)
- Existen **dos tipos de parámetros** que podemos pasar a una función: por valor (se crea una copia del objeto cuando se ejecuta la función) y por referencia (se pasa la referencia a la posición de memoria donde está guardado el objeto y por tanto las modificaciones hechas sobre él se reflejan también al retornar de la función).
- La sintaxis es distinta para estos dos tipos de parámetros, así:
  - **Parámetros por valor:** `tipo_param nombre_param`.
  - **Parámetros por referencia:** `tipo_param & nombre_param`.
- Los parámetros de una función pueden ser de cualesquiera de los tipos admitidos por el lenguaje **excepto vectores o matrices**.
- La expresión de retorno es cualquier expresión permitida con el mismo tipo que aquel con el que definimos la función (podría incluso ser una llamada a otra función del mismo tipo).

- **Instrucción de declaración de procedimientos:** Al igual que en las funciones, generamos un código que no se ejecutará hasta que sean llamados. Su sintaxis es la siguiente:

```
proc nombre_proc (tipo_1 param_1,...,tipo_n param_n){
    ...
    cuerpo_procedimiento
    ...
}
```

Todas las observaciones sobre los parámetros hechos antes para las funciones son igualmente válidas para los procedimientos.

- **Instrucción de llamada a procedimientos:** Indicamos que queremos ejecutar el cuerpo de un procedimiento declarado previamente indicándole los argumentos con los que queremos llamarle. Su sintaxis es la siguiente:

```
    call nombre_proc (arg_1,...,arg_n);
```

Los argumentos tienen que ser igual en número y en tipo a aquellos parámetros con los que declaramos el procedimiento y además se comprobará que para los **parámetros por referencia** recibamos expresiones a las que se le pueda asignar un valor (identificadores, posiciones de vectores, campos de estructuras..).

4. Expresiones

Nuestro lenguaje proporciona para las expresiones los operadores usuales para las operaciones aritméticas y booleanas, cuyas prioridades y asociatividades son las usuales en los lenguajes de programación y quedan claras en la gramática. Hacemos algunas observaciones:

- Como ya hemos dicho, sólo unas pocas expresiones pueden aparecer en la **parte izquierda** de asignaciones. Estas son los **identificadores de variables**, el **operador punto** (acceso a campos de estructuras), el **operador corchetes** (para acceder a posiciones de vectores) y el **operador asterisco unario** (para acceder a la variable apuntada por un puntero).
- En la **parte derecha** de las asignaciones y en cualquier otro sitio donde pueda aparecer una asignación (siempre que los tipos cuadren) permitimos cualquier expresión, incluyendo evidentemente las que podían usarse en las partes izquierdas.
- Para distinguir las expresiones que pueden aparecer en partes izquierdas todas las expresiones tienen un **atributo booleano** al que damos valor al construir el árbol y que nos permitirá hacer esta distinción en la comprobación de tipos de la parte del análisis semántico.
- Los **parámetros por referencia** de funciones y procedimientos sólo pueden recibir como argumentos en sus llamadas expresiones que podrían ir en la parte izquierda de una asignación, por lo que usaremos también este booleano del que hablábamos para comprobar este hecho en la comprobación de tipos.
- Las llamadas a funciones (con las mismas condiciones para los argumentos que indicábamos en la instrucción `call`) sólo pueden aparecer en partes derechas de asignaciones (y demás lugares en los que puedan aparecer estas expresiones). Esto implica que **no podemos llamar una función si no forma parte de una expresión derecha**.
- Podemos consultar el **tamaño de un vector**, ya sea estático o dinámico mediante la instrucción `.size`. Esta instrucción se aplica sobre expresiones de tipo vector (podrían aparecer puntos y corchetes en ella) y nos devuelve un entero con la dimensión del vector. En el caso de vectores con más de una dimensión devolvemos el tamaño en la primera dimensión. Para obtener las dimensiones más internas accedemos con corchetes(con

cualquier índice permitido) a submatrices más internas y devolvemos el tamaño en esa dimensión. El siguiente ejemplo aclara esto:

```
vector<vector<vector<int>> v =
    creaVector(creaVector(creaVector(0, 3), 4), 5);
int t1 = v.size
int t2 = v[0].size
int t3 = v[0][0].size

# Tendriamos los valores: t1 = 5, t2 = 4, t3 = 3
```

5. Bloques y ámbitos

Nuestro lenguaje cuenta con un **ámbito principal**, de nivel 0, que es el fichero fuente sobre el que escribimos nuestro código. Las variables, tipos y funciones aquí definidos serán visibles en todo el programa y en cierto modo actúan como **variables globales** a nuestro programa. Así se abre un nuevo ámbito de definición de variables en los siguientes casos:

- En el cuerpo de un bucle.
- En el cuerpo de la parte del if de una isntrucción condicional.
- En el cuerpo de la parte del else de una isntrucción condicional con else.
- En el cuerpo de una función.
- En el cuerpo de un procedimiento.
- En cada uno de los case de un switch.

Así, las variables, tipos, funciones y procedimientos declarados en uno de estos bloques **sólo serán accesibles dentro del bloque**. Podemos anidar estos bloques del modo que queramos teniendo en cuenta las restricciones de visibilidad de las declaraciones que hagamos dentro. Esto nos da gran libertad en la definición de **procedimientos y funciones anidadas** pues podemos declararlas en cualquier parte de nuestro programa.

Por otro lado tenemos los **marcos de activación** que sólo se crean al declarar funciones y procedimientos y son los que determinan las direcciones que damos a variables. Así, la **profundidad de anidamiento** (a la que nos referimos en la implementación del compilador como "pa") sólo cambia al abrir un nuevo bloque de procedimiento y función. Esto tiene sentido pues sólo al ejecutar el cuerpo de una función o un procedimiento cambiamos el valor de `MP` y sólo aquí deberíamos volver al valor inicial de las direcciones para asignárselas a las nuevas variables declaradas sin machacar lo que ya tuviésemos guardado. Así todo lo declarado directamente en el fichero fuente tiene `pa = 1`, lo declarado dentro de una función declarada en el fichero fuente tiene `pa = 2` y así sucesivamente, mientras que aquello declarado dentro de cualquier otro bloque que

no sea de procedimiento o función sigue teniendo el mismo `pa` que tenía antes. Así, por ejemplo, una variable declarada dentro de un if que a su vez está en el código fuente sigue teniendo `pa = 1`.

6. Compilación de código fuente

La compilación y generación de código sigue las siguientes fases, construyendo las estructuras necesarias y detectando los posibles errores en cada una de ellas:

- Análisis léxico
- Análisis semántico
- Análisis semántico
- Generación de código

Detallamos un poco más las dos últimas fases que son las que se nos pedía implementar en esta última parte de la práctica.

6.1. Análisis semántico

Esta parte, a la que se supone que hemos llegado sin errores léxicos y sintácticos, consta a su vez de dos fases:

- **Vinculación:** Hacemos un recorrido por el árbol de sintaxis abstracta construyendo una **tabla de símbolos** en la que guardamos las variables, funciones, procedimientos y tipos declarados en cada bloque y que por tanto son visibles en el mismo y comprobamos que todas las apariciones de identificadores se corresponden con identificadores declarados y visibles en ese nodo del árbol. Además, para estas apariciones nos guardamos en el nodo del árbol la **referencia** de la instrucción dónde fueron declarados para tener información de este identificador, entre otras cosas, de su tipo. Es en esta parte donde se sustituyen los alias por el tipo primitivo al que se refieren y se sustituyen los enumerados por los enteros correspondientes.
- **Comprobación de tipos:** Hacemos un segundo recorrido por el árbol de sintaxis abstracta comprobando que los tipos de las expresiones que aparecen en cada instrucción son los correctos. Aquí comprobamos también además todas las **restricciones que habíamos impuesto en algunas instrucciones** que no se habían comprobado antes, como por ejemplo que en la declaración de un vector sólo se use `creaVector` o que los argumentos en llamadas para parámetros pasados por referencia sean expresiones asignables.

6.2. Generación de código

Esta parte de la compilación se realiza también en **dos fases** que implican dos pasadas nuevas por el árbol de sintaxis abstracta.

- **Asignación de direcciones de memoria:** Realizamos una primera pasada por los nodos del árbol en la que construimos la función ρ de la que se habla en los apuntes de la asignatura y que nos da para cada variable la dirección de memoria en la que se almacenará. Así, vamos abriendo bloques al igual que lo hacíamos para la vinculación y rellenamos la información correspondiente para tener acceso luego en la fase de generación de código propiamente dicha. Así, en cada **bloque** almacenamos la siguiente información que después necesitaremos recuperar de un modo u otro:

  - **Mapa de identificadores de variable a direcciones de memoria** asignadas para los mismos (para identificadores de variables de tipo vector o registro se indica la dirección de comienzo de la zona de memoria asignada para los mismos.
  - **Mapa de identificadores a tamaños:** para aquellas variables de tipos no básicos se guarda el tamaño que ocupan. Para vectores dinámicos guardamos el tamaño de su parte estática que sí que es algo de lo que tenemos información en tiempo de compilación.
  - **Mapa de identificadores de vectores a dimensiones estáticas:** para los vectores estáticos almacenamos el tamaño por cada una de sus dimensiones para luego poder comprobar en tiempo de ejecución que no nos salimos de rango en los accesos.
  - **Mapa de identificadores de vectores a booleanos** que indican si, dado un identificador de vector, este es estático o no.

  Adicionalmente almacenamos el **tamaño del bloque**, el **tamaño de la zona de variables en memoria** declarada en ese bloque (para la instrucción `ssp`), una **referencia al bloque sobre el que nos han declarado** como bloque (si `padre == null` estamos declarados directamente sobre el fichero fuente) y **la dirección** por la que vamos asignando.

- **Generación de código propiamente dicha:** En esta fase generaremos un fichero de texto (`.txt`) con la traducción a código máquina de las instrucciones en nuestro lenguaje indicadas en el fichero de código fuente. Para ello recorremos el árbol y generamos las instrucciones en código máquina correspondientes a cada instrucción de nuestro lenguaje y las guardamos en una **lista de instrucciones** (que permitirá más fácilmente añadir información sobre etiquetas de saltos y similares que conoceremos más tarde) y una vez terminado el proceso **volcamos estas instrucciones en el fichero** `.txt` antes mencionado.

Para generar el código para nuestras instrucciones hemos seguidos las indicaciones dadas en los apuntes de generación de código, adaptándolas de manera

conveniente a nuestro lenguaje y las estructuras de datos que nos habíamos construido pero la idea final viene a ser la misma.

Pasamos a continuación una de las partes opcionales que hemos implementado y en la que nos hemos ligeramente de lo que en los apuntes se muestra: **los vectores y matrices dinámicos.**

En primer lugar consideramos como **vector dinámico** todo aquel para que en tiempo de compilación no podamos saber el tamaño en alguna de sus dimensiones (porque sean una expresión que no sea un entero como tal). Para identificarlos vemos si en la declaración hay alguna dimensión para la que no indicamos el tamaño con un **entero** como tal. Una vez hecho esto y guardada la información correspondiente en el bloque en la fase de asignación de memoria **el código se genera de manera distinta dependiendo del tipo de vector** que se trate.

Así, para **vectores estáticos** reservamos tantas posiciones de memoria como tamaño ocupe el vector (es algo que podemos calcular estáticamente) y para **vectores dinámicos** reservamos una posición en la que guardaremos la **dirección dinámica** del vector, otra con el **tamaño total** y una por cada dimensión en la que guardaremos el **tamaño para esa dimensión**. En total un vector dinámico de k dimensiones ocupa $k + 2$ posiciones en la parte de la pila de variables (estática).

Así, en las **declaraciones de vectores** generamos código de dos maneras distintas dependiendo del tipo de vector:

- Para **vectores estáticos** cargamos en cada una de las posiciones de memoria reservadas el valor resultante de evaluar (generar código) para la expresión indicada como valor inicial.
- Para **vectores dinámicos**:
  - Vamos generando código para las **expresiones de cada una de sus dimensiones** y multiplicándolas (todo esto con instrucciones de la máquina-P) y guardamos en la **segunda posición** que nos habíamos reservado este valor.
  - Una vez hecho esto ya sabemos cuanta memoria tenemos que **reservar** y hacemos un `new` (de la máquina-P) para reservar tantas posiciones como tamaño acabamos de calcular y guardamos esta dirección de memoria dinámica en la **primera posición** estática que habíamos reservado.
  - A continuación guardamos, **a partir de la tercera posición** que nos habíamos reservado y hasta agotar el número de dimensiones del vector, el **tamaño por cada dimensión**.

- Por simplicidad **es necesario asignar valor inicial a los vectores dinámicos** aunque en realidad y dada la complicación que suponía y la poca información de la que disponíamos para ello optamos por **no generar código para este valor inicial** y no ser llega a cargar nunca. La única manera ahora es ir asignando valor posición a posición si queremos dar valor a los elementos del vector.

Por otro lado recalcamos que **no hemos permitido pasar vectores como parámetro a funciones y procedimientos** porque en la declaración de los mismos deberían considerarse como vectores dinámicos (es imposible que sepamos el tamaño en cada una de las dimensiones) y además tampoco se nos indica bien como hacer la copia de los valores a un vector dinámico.

Sin embargo esto tampoco supone mucha limitación a nuestro lenguaje pues conviene recordar que **tenemos acceso a las variables declaradas en el mismo bloque que la función** (por lo que podemos acceder a vectores declarados fuera dentro de la función y evidentemente nos podemos declarar nuestros propios vectores dentro de la función).

Lo que sí que podemos pasar como argumento a una función son **estructuras** pues de estas sí que conocemos su **tamaño** estáticamente.

7. Gestión de errores

7.1. Recuperación de errores sintácticos

La gramática que reconoce que el programa introducido por el usuario sea sintácticamente correcto además incluye varias producciones de error que permiten recuperarse de los mismos, indicándose un símbolo terminal a partir del cual intenta reconocer *tokens* válidos para dicha recuperación.

Esta recuperación de errores se ha hecho a nivel de instrucción, pues al fin y al cabo un programa de nuestro lenguaje es una sucesión de instrucciones. Y por tanto, un error en, por ejemplo, una expresión, queda se produce en particular sin poderse reducir una producción de instrucción.

Para hilar más fino en estos errores, se han ido reconociendo progresivamente distintos tokens, en varias producciones, antes de colocar la palabra *error*. Así, por ejemplo, en una declaración de función podemos determinar si hay una error en . Este en concreto se ha hecho como se puede ver en la figura 1

Además, en caso de que la recuperación del último error no sea posible porque se encuentre el final de fichero antes de recuperarse de él, se informa pertinentemente al usuario y se solventa cogiendo la excepción java que se lanza. Además se lleva una cuenta de la cantidad de errores sintácticos encontrados que se muestra al usuario al finalizar esta fase y en caso de exceder los 15 errores

```
InsFun ::= FUN error LLAVESCIERRE
    {: System.err.println("Error en el tipo de declaración de la función \n");
     GestionErroresTiny.numErroresSintacticos++;  RESULT = null; :};

InsFun ::= FUN Tipo error LLAVESCIERRE
    {: System.err.println("Error en el nombre de la funcion \n");
     GestionErroresTiny.numErroresSintacticos++;  RESULT = null; :};

InsFun ::= FUN Tipo IDEN error LLAVESCIERRE
    {: System.err.println("Error en los parámetros de la función \n");
     GestionErroresTiny.numErroresSintacticos++;  RESULT = null; :};

InsFun ::= FUN Tipo IDEN PAP Parametros PCIERRE error LLAVESCIERRE
    {: System.err.println("Error en el cuerpo de la función \n");
     GestionErroresTiny.numErroresSintacticos++;  RESULT = null; :};
```

Figura 1: Ejemplo de producciones de error colocada en la gramática

se para la compilación al entenderse que el programa está bastante mal y que el usuario tiene mucho que corregir.

7.2. Recuperación de errores semánticos

En caso de que haya habido errores sintácticos se ha decidido no pasar a la fase de análisis semántico. Con un AST incompleto respecto al programa que el usuario intentaba compilar es complicado ser coherente en los errores venideros. Muchos errores semánticos serían debidos a problemas sintácticos previos. Es mejor que el usuario arregle primero los otros errores, antes de pasar a esta fase.

Eso sí, dentro de los errores semánticos hay dos subfases de errores, los de vinculación y comprobación de tipos se hacen siempre. Aunque haya habido errores de vinculación (variables, procedimientos, tipos del usuario usados pero no declaradas), se pasa a la comprobación de tipos, intentando en la medida de lo posible ignorar las instrucciones que tienen ya un fallo para no mostrar errores equívocos que puedan confundir al usuario.

En general los errores, sobretodo en comprobación de tipos, son muy precisos e indican el problema concreto que está sucediendo. Por ejemplo, si una llamada a una función falla, indica si es porque el número de argumentos es incorrecto, porque el tipo de algún argumento es incorrecto, o porque el tipo que devuelve la función no cuadra con el tipo con el que se pretende usar.

Al igual que como con los errores sintácticos se lleva una cuenta de los errores semánticos encontrados y se informa al usuario. Y por supuesto no se pasa a la fase de generación de código habiendo errores en esta fase. Del mismo modo se hace con los errores semánticos en cso de exceder los 15. Se para la compilación.