UNIVERSIDAD COMPLUTENSE DE MADRID

Entrega Final

PROCESADORES DEL LENGUAJE

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

Realizado por: Ángeles Plaza Gutiérrez y Ana Belén Duarte León

1. INTRODUCCIÓN

Durante todo el cuatrimestre en la asignatura de Procesadores del Lenguaje hemos desarrollado un lenguaje parecido a C/C++ y un compilador para poder probarlo. Los aspectos principales que destacar de nuestro lenguaje son los siguientes:

- Cada programa se escribe en un único fichero fuente.
- La función main es el código de este fichero fuente, pues al ejecutarlo las instrucciones se van ejecutando en el orden en el que aparezcan dentro del mismo.
- Se permiten comentarios de una línea que empiezan por // y serán ignorados en la ejecución.
- Solo se generará código para los tipos booleanos y los enteros.
- Se reconocerán y se capturarán una serie de errores que detallaremos más adelante, tanto léxicos, como semánticos
- Se proporcionarán adjuntos a esta memoria, archivos txt que permitirán probar las funcionalidades principales de esta práctica.

2. IDENTIFICADORES Y ÁMBITOS DE DEFINICIÓN

● Declaración de variables de tipos simples

Vamos a declarar variables con valor y sin valor inicial.
La declaración de variables sin valor inicial se hará de la forma: *tipo nombre*;

La declaración de variables con valor inicial se realizará de la forma: *tipo nombre = valor*;

```
//declaración de variables
int x;
int y = 6;
int num_caminos = 0;
```

● Comentarios

Podremos escribir comentarios de una sola línea. Los comentarios se escribirán precedidos de //. Lo que se escriba en esa línea tras // será ignorado.

```
// comentario de una línea
```

- **Arrays unidimensionales**

Los arrays unidimensionales pueden ser también declarados sin valor inicial o con valor inicial. No se podrá asignar la declaración de un array a una variable, la instrucción deberá ir sola.
Los arrays sin valor inicial son de la forma *tipo nombre [tamaño]*;
Los arrays además pueden tener todas las posiciones inicializadas con un mismo valor y se hace de la forma: *tipo nombre [tamaño {valor}]*;
No se puede dar valor inicial a un array asignándole otro array aunque sean ambos del mismo tipo, la única declaración válida para arrays es la mostrada en la siguiente figura.

```
int dias[4];
bool array[10];
int numero[4 {10}];
```

- **Arrays multidimensionales**

Los arrays multidimensionales pueden ser también declarados sin valor inicial o con valor inicial como en el caso de los unidimensionales.
La diferencia es que en nuestros corchetes podemos poner tantos números como dimensiones tengamos : tipo nombre [3, 4, 5 {7}].

- **Bloques anidados: WHILE IF**

Los bloques anidados los declararemos como en el lenguaje de C++, con la excepción de que se deberán poner siempre llaves, aunque haya una única instrucción. Las condiciones de if y de while irán entre paréntesis.
Podremos tener while, if y for anidados.

```
//bloques anidados
while(ok != false){
   if(x > 0){
       x = x-3;
   }
   //bucle for
   for(int i = 0; i < 20; ++i){
        x = 3+i;
   }
}
```

- **Funciones**

Comienzan con la palabra ***fun*** seguido del tipo de la función. A continuación enviamos los argumentos como parámetros por valor entre paréntesis y separados por comas. Por último aparece el cuerpo de la función entre llaves con la instrucción ***return***.

La expresión que se devolverá en return será del mismo tipo que tenga la definición de la función al principio.

Las funciones no podrán recibir arrays como parámetros.
fun tipo nombre(argumentos){ instrucciones;}

```
//declaración de una función
fun bool check(int p1, int p2){
   if(pi > p2){
       return true;
   }
   else{
       return false;
   }

}
```

- **Procedimientos**

Los procedimientos se definirán de la forma
proc void nombre(argumentos){instrucciones}.

Proc es la palabra reservada para los procedimientos, en este caso no tendrán return.

Los procedimientos no podrán recibir arrays como parámetros

```
//declaración de un procedimiento
proc void suma(int n){
   n = n+7;
}
```

3. TIPOS

- **Tipos básicos: int, bool, char, float**

Se definirán como en el lenguaje C++. Las palabras true y false se reservarán para el tipo bool. Los elementos de tipo char aparecerán entre comas simples 'x' y los elementos de tipo float usan un punto (.) antes de su parte decimal.

```
int num = 10;
bool ok = true;
char letra = 'b';
float num2 = 45.7
```

- **Operadores infijos**

Estos operadores tanto aritméticos como relacionales y lógicos presentarán las prioridades y asociatividades usuales. Veamos cuáles son tipo a tipo.

-Operadores unarios: !, +,-

-Operadores aritméticos: +, -, *, /, %

-Operadores relacionales: >, <, >=, <=, ==, !=

-Operadores lógicos: &&, ||

```
//ejemplos con operadores de distinto tipo
int x = 4+3;
int y = 8/2;
bool z = true && ok;
if(x >= 7){
   x = x-5;
}
```

- **Tipos con nombre: STRUCT**

El tipo Struct se declara así: *struct nombre* { };

En el interior de los { } podemos declarar campos de diferentes tipos. Para acceder a un campo concreto emplearemos la sintaxis nombreStruct.campo;

Además, nuestros structs son muy especiales. Podemos pensarlos como elementos únicos que almacenan información, ya que solo habrá una instancia de cada struct, la de la propia declaración. Esto además nos posibilita asignar un valor inicial por defecto a los campos de un struct si lo deseamos al declararlo. Podemos declarar un struct con valores en algunos campos y en otros no. Por tanto el nombre asignado a un struct será un identificador único y nos permitirá acceder a los campos de ese struct.

Sin embargo, mediante el acceso a los campos del struct podemos asignar valor a un campo o dar el valor del campo a una variable del mismo tipo.

Esta estructura junto a enum serán las únicas que podrán recibir nombres de tipos.

No será posible declarar un struct de forma directa en una variable si éste no está definido antes.

```
struct empleado{
   int id = 5;
   bool encargado;
}
int n = empleado.id;
```

- **Definición de tipos por el usuario: ENUM**

Podemos declarar además tipos enumerados que se escriben de la forma: *enum nombre = {valor_1|valor_2...}*;

Para acceder a los elementos de enum emplearemos ‘:’

```
enum color = {ROJO | VERDE | AZUL | NARANJA};
Coloc camiseta = (Color:ROJO);
```

Como se menciona en la última parte del documento, no hemos realizado el tipado de los enums, por lo que a la hora de probar la práctica esta última

frase daría error. Sin embargo hemos implementado todo lo anterior: léxico, sintáctico y vinculación.

4. CONJUNTO DE INSTRUCCIONES DEL LENGUAJE

- **Instrucciones de asignación de variables**

Las realizaremos como en C++.

```
int v;
v = 0;
```

Según la gramática solo un tipo determinado de expresiones pueden aparecer en la parte izquierda de una asignación. Estas expresiones se llaman **designadores** y constan de: identificadores de variables, el operador de acceso a campos structs y el operador corchetes que nos permite el acceso a arrays. En la parte derecha de la asignación permitimos cualquier expresión que sea del mismo tipo que el de la variable.

Debemos tener en cuenta que no permitimos la asignación de arrays completos, pero sí podemos asignar una valor a un elemento de un array o asignar el valor del elemento de un array a una variable, siempre y cuando tipen correctamente.

Recordamos que en nuestro programa los struct aparecerán al comienzo del mismo, por lo que tampoco podremos hacer asignaciones del tipo struct a = b; siendo b un struct ya declarado.

De la misma manera que con los arrays, podemos asignar a una variable el valor del campo de un struct y viceversa.

- **Instrucciones de arrays**

```
array[i] = x;
```

para dar un único valor.

- **Instrucción print:**

Podremos emplear la instrucción print para mostrar por pantalla una expresión: print(exp);

- **Condicional con una, dos o varias ramas**

-De una rama:

```
if(condición){

 }
```

Donde { } contiene el cuerpo que se ejecuta si se cumple la condición.

-De dos ramas:

```
if(condición){

 }
 else {

 }
```

Donde los { } contienen el cuerpo que se ejecuta si entramos por esa rama.

-De más de dos ramas:

```
if(condición){

 }
 elif(condición){

 }
 ...
 elif(condición){

 }
 else{

 }
```

Donde los { } contienen el cuerpo que se ejecuta si entramos por esa rama.

- **Bucles: FOR Y WHILE**

Bucle while: tiene la siguiente estructura

```
while(condicion){
  cuerpo...
}
```

El cuerpo del while estará formado por una lista de instrucciones de nuestro programa que se ejecutarán siempre que se cumpla la condición del bucle.

Bucle for: tiene la siguiente estructura

```
for(int i = valorInicial; cond; paso){
   cuerpo...
}
```

Como en el caso de while, el cuerpo del for será una lista de instrucciones que se ejecutarán mientras que se cumpla la condición del for. Según se ha definido la gramática el for siempre tiene que comenzar con una declaración con valor inicial de la variable que controlará el número de iteraciones del bucle. Cond será una condición booleana que se evaluará para decidir si el bucle acaba o se realiza otra iteración. Por último, paso será una expresión que se corresponderá con un ++i o --i y nos permitirá cambiar la variable que controla la ejecución del bucle.

- **Expresiones formadas por constantes**

Las expresiones pueden ser valores constantes, también tenemos true y false como constantes booleanos.

```
bool ok = enc && true;
//true es constante en la expresión

num = num*4;
```

- **Identificadores con y sin subíndices**

Los identificadores empezarán por una letra ; y podrán contener no, letras, números y _.

```
//identificadores válidos
int num_Cajas ;

bool familiade5;
```

- **Llamadas a funciones**

Usaremos la palabra reservada call seguida del identificador de la función o procedimiento al que queremos llamar seguido de los argumentos con los que queremos llamarle entre paréntesis. Los parámetros se pasan por valor y deben coincidir en número y tipo con los de la declaración de la función o procedimiento.

```
//con la función definida anteriormente
int x = 4;
call check(x);
```

Además, podremos asignar a una variable el valor devuelto por una función:

```
int y = 3;

int x = call suma(y);
```

Donde suma es una función que devuelve un entero. Sin embargo, no permitimos las llamadas a función con elementos que no sean variables, call suma(3) daría error.

5. GESTIÓN DE ERRORES

Se indicará el tipo de error y la fila y columna donde se encuentra mediante un mensaje al usuario. Se detendrá la compilación. La proseguiremos para detectar otros errores en la medida de lo posible.

- **Errores sintácticos**

La gramática asegura que el programa introducido por el usuario en el archivo de código fuente es sintácticamente correcto. Se han incluido varias producciones de error para poder recuperarse de los mismos y seguir detectando otros posibles errores pero hasta un máximo de 10. En el momento que la gramática encuentra y "trata" 10 errores sintácticos se detiene la compilación puesto que se entiende que el programa escrito tiene demasiados errores y el usuario debe corregirlo.

La recuperación de errores se ha hecho a nivel instrucción puesto que son las funciones y sus cuerpos de instrucciones componentes de los programas. En particular, vamos reconociendo tokens adecuados de una instrucción e incluimos la palabra error cuando queremos reconocer un error en un momento concreto. Por ejemplo en la declaración de una función reconocemos varios tipos de errores: un error en la declaración de la función, que puede generarse por ejemplo si incluimos caracteres no válidos para un identificador, error en los argumentos de la función, o error en el cuerpo de la función tanto si ésta recibe parámetros como si no lo hace. Veamos el código:

```
InsFun      ::= FUN Tipo error PAP PCIERRE Cuerpo
{:System.err.println("Error en el identificador de la función \n");
RESULT = null;  :};
InsFun      ::= FUN Tipo IDEN PAP error PCIERRE Cuerpo
{:System.err.println("Error en los parámetros de la función \n");
RESULT = null;  :};
InsFun      ::= FUN Tipo IDEN PAP PCIERRE error
{:System.err.println("Error en el cuerpo de la función \n");
RESULT = null;  :};
InsFun      ::= FUN Tipo IDEN PAP LParam PCIERRE error
{:System.err.println("Error en el cuerpo de la función \n");
RESULT = null; :};
```

- **Errores semánticos**

Se ha decidido que si existen errores sintácticos en el código fuente no se pasará a hacer la comprobación de la existencia de errores semánticos puesto que un programa con errores en la gramática generará demasiados problemas a la hora de estudiar los errores semánticos.

El estudio de errores semánticos tiene dos fases: la vinculación y el tipado. En este caso, primero se realiza la vinculación y aunque haya errores se llevará a cabo, a continuación, la comprobación de tipos. Las variables, procedimientos, accesos a structs o funciones que no sean hayan sido

declaradas se reportarán en la vinculación y serán ignoradas a la hora de comprobar el tipo puesto que no queremos arrastrar este error que ya hemos detectado. En caso contrario, se mostrarían errores de tipos que podrían confundir al usuario.

En la comprobación de tipos los errores detectados son de una mayor precisión indicando dónde está el problema concretamente al usuario. Por ejemplo, si llamamos a una función con unos argumentos cuyo tipo no se corresponde al tipo de los argumentos que hay en la declaración de la función, nuestro compilador será capaz de reportar este hecho y decirle en qué fila del código se ha realizado esta llamada errónea. Otro ejemplo de error que nos devuelve la comprobación de tipos es que hagamos un return de una expresión con un tipo distinto del que se había declarado la función.

Como ocurre con los errores sintácticos, llevaremos una cuenta de cuántos errores semánticos se han detectado y se informa al usuario por pantalla de cada uno de ellos. En este caso, permitimos un máximo de 15 errores semánticos antes de detener la compilación.

Como es obvio, solo en caso de que el código no presente errores semánticos se procederá a la generación de código.

```
public void chequea() {
        Iden id =  (Iden) iden;
        nodoFun = TabladeSimbolos.getDeclaracion(id.getId());//obtengo el nodo de esta funcion
        if (((Ins) nodoFun).tipoIns() == TIns.FUN && ((InsFun) nodoFun).getNumParametros() == numArgs) {
            boolean ok = true;
            List<Parametro> params = ((InsFun) nodoFun).getParametros();
            for (int i = 0; i < numArgs; ++i) {//debo comprobar que el tipo de los parametros y de los argumentos coincide
                TTipo tipoParam = params.get(i).getTipo().tipoTipo();
                Iden idArg = (Iden) argumentos.get(i);
                ok =  (idArg.getTipo() == tipoParam);
                if(!ok){
                    GestionErroresExp.errorSemantico(fila, columna, "El tipo de uno de los argumentos no coincide con el de la función");
                }
            }
        }
        else if (((Ins) nodoFun).tipoIns() == TIns.PROC && ((InsProc) nodoFun).getNumParametros() == numArgs) {
            boolean ok = true;
            List<Parametro> params = ((InsProc) nodoFun).getParametros();
            for (int i = 0; i < numArgs; ++i) {//debo comprobar que el tipo de los parametros y de los argumentos coincide
                TTipo tipoParam = params.get(i).getTipo().tipoTipo();
                Iden idArg = (Iden) argumentos.get(i);
                ok =  (idArg.getTipo() == tipoParam);
                if(!ok){
                    GestionErroresExp.errorSemantico(fila, columna, "El tipo de uno de los argumentos no coincide con el de la función");
                }
            }
        }
        else{
            GestionErroresExp.errorSemantico(fila, columna, "El número de argumentos es erróneo.");
        }
    }
```

6. VINCULACIÓN y TIPADO:

Como hemos dicho anteriormente, el análisis semántico del programa solo se llevará a cabo en caso de no haber detectado ningún error sintáctico. Por tanto los pasos que sigue nuestro programa son los siguientes:

1. Análisis Léxico
2. Análisis Sintáctico
3. Análisis Semántico
4. Generación de Código

Vinculación: Para realizar la vinculación recorremos el árbol de sintaxis abstracta (AST) creando la tabla de símbolos. En ella introducimos funciones, procedimientos y variables declarados en cada bloque y que son visibles en el mismo. Como sabemos en nuestro programa los enumerados y structs se declaran al comienzo, por lo que para poder acceder a ellos a lo largo de la ejecución del programa, debemos mantener abierto su bloque hasta el final. Cuando en el programa se haga referencia a un identificador comprobaremos que está declarado y es visible dentro de su bloque ayudándonos de la tabla de símbolos. Además, donde sea necesario nos guardaremos una referencia al nodo insertado en la tabla para tener información, por ejemplo, sobre su tipo.

Las directrices que hemos tomado para vincular son las siguientes: cuando tenemos anidación de bloques, únicamente permitiremos declarar variables que no estén presentes en bloques externos. Por ejemplo:

```
for( int i = 5; i < 5; ++i){
  int j = 3;
  bool casa = true;
  if(casa){
     int j = 0;
     int k = 4 + 7;
  }
}
```

En este caso, habíamos declarado la variable j en el for. Por tener un if anidado que también declara esta variable j, se nos mostraría un error: ERROR SEMÁNTICO: *fila: x columna: y. El identificador ya existe. Tomamos la primera declaración.*

Comprobación de tipos: volvemos a recorrer el AST comprobando ahora que los tipos de las expresiones que aparecen en cada instrucción son correctos.Además es en este momento cuando realizamos otras comprobaciones pertinentes para el correcto funcionamiento de nuestro

programa como por ejemplo, que las funciones no pueden recibir arrays como parámetro. A la hora de realizar el tipado, hemos tenido problemas con los enum, por lo que finalmente hemos decidido omitirlos en esta última parte de la práctica: tipado y webassembly. Sin embargo, hemos realizado lo necesario para implementarlos en el resto de aspectos.

7. WEBASSEMBLY:

A continuación exponemos qué cosas hemos llegado a implementar en Webassembly.

En primer lugar, como hemos comentado anteriormente, no implementaremos nada sobre enums. A esto se suman también los structs, arrays y llamadas a funciones, que quedarán al margen de la generación de código.

Por otro lado, sí hemos implementado tanto asignaciones como declaraciones. Asimismo, todo tipo de operaciones binarias y unarias, bucles while y for. En lo referente a la instrucción if, podemos generar código de instrucciones que solo tengan if o que tengan if & else, así como de aquellas que tienen una o más condiciones de elif.

En cuanto a la reserva de memoria, hemos implementado funciones en cada una de las clases del AST que calculan cuánto ocupa ese elemento en concreto: **maxMemoria**. De esta manera, al llamar a la función del programa principal que calcula el tamaño máximo, esta función a su vez pide el tamaño máximo de cada función o procedimiento que contiene, y así sucesivamente hasta que le llega el retorno al programa principal, que ya tendrá la máxima memoria que necesita gracias a subcálculos de la máxima memoria de cada elemento. Recordamos que esta memoria hace referencia a declaraciones. A la hora de reservar memoria, será el retorno de maxMemoria*4 (recordamos que estamos trabajando con bytes) + (2*4), esto último procede del espacio para almacenar sp y pl.

También aparece la función **generaEtiqueta** que nos permite asignar a cada declaración una etiqueta que indica su dirección. Debemos hacer notar que cuando cerramos un bloque, desaparecen todas las etiquetas asignadas a él, por lo que la cuenta continua desde el número anterior al bloque en cuestión. Esta etiqueta también va asociada al tamaño, por lo que si queremos guardar un array de tamaño x, la siguiente le asignamos el valor de la última etiqueta +x. Hacemos notar que en nuestro caso estas etiquetas son bytes, por lo que van de 1 en 1. A posteriori deberemos multiplicar este valor de etiqueta *4 como hicimos con la memoria.