Lenguajes de Programación y Procesadores de Lenguajes

Construcción de un compilador para: `MenosC`.18 2017-18

1. Introducción y justificación del proyecto

Las practicas de la asignatura de *Lenguajes de Programación y Procesadores del Lenguaje* están orientadas a la realización de un proyecto –*construcción de un compilador*– donde el alumno pueda poner en práctica los conocimientos del funcionamiento de un compilador aprendidos en las sesiones de teoría. La motivación que justifica esta elección metodológica se puede resumir:

1. Conseguir que el alumno adquiera una visión más realista del funcionamiento de un compilador.
2. Facilitar la comprensión de algunos conceptos que difícilmente se entenderían solo en las sesiones teóricas.
3. Constatar que el alumno habrá participado activamente en la elaboración de un compilador real.
4. Incidir en que esta alternativa es la más cercana al tipo de trabajos que el ingeniero en informática se va a encontrar en sus labores profesionales.

Para evitar, en lo posible, la sobrecarga de trabajo que un proyecto de esta envergadura conlleva se han tomado una serie de medidas correctoras:

1. Impulsar que el proyecto se realice en pequeños grupos (como máximo cuatro alumnos). Además, con ello se consigue fomentar las habilidades del trabajo en equipo requisito imprescindible en todo ingeniero informático.
2. Proporcionar a los alumnos un adecuado material de ayuda que les permita reducir significativamente el trabajo de codificación para centrarse en los problemas típicos de la construcción de compiladores.
3. Planificar un conjunto de seminarios, en grupos reducidos, para la descripción pormenorizada del material y herramientas específicas del proyecto.
4. Reforzar las tutorías en el laboratorio. La labor del tutor no solo será la de resolver las dudas y problemas planteados sino también la de sugerir mejoras, detectar problemas, motivar hábitos de trabajo en equipo y enseñar a generar y documentar buenos programas.

1.1. Presentación y objetivos

El objetivo principal es la *construcción de un compilador* completo para un lenguaje de programación de alto nivel, sencillo pero no trivial, al que denominaremos `MenosC`. El lenguaje elegido, `MenosC`, es un lenguaje basado en el lenguaje `C`, con algunas restricciones de tipos del lenguaje `C++`.

Para facilitar la tarea de implementación y verificación del proyecto, éste se divide en tres etapas:

`Parte I`	Construcción del analizador léxico-sintáctico
`Parte II`	Construcción del analizador semántico
`Parte III`	Construcción del generador de código intermedio

Se recomienda al alumno que, además de estudiar detenidamente este documento, consulte los manuales en linea de FLEX y BISON que se puede encontrar en la plataforma `PoliformaT` (menú recursos > material para prácticas).

1.2. Material de prácticas

A lo largo de las tres partes en las que se divide el desarrollo del compilador se proporcionará diverso material de prácticas para que sirva de ayuda en la tarea de codificación del proyecto. Tanto si se trabaja en los laboratorios como desde casa –mediante una sesión remota RDP ("Remote Desktop Protocol")[1]– este material se puede encontrar en:

`asigDSIC/ETSINF/lppl/`

Independientemente de que se pueda trabajar en casa con computadores propios, es importante advertir que el código del compilador debe funcionar para la distribución instalada en los equipos de los laboratorios docentes.

1.3. Evaluación

La evaluación de las prácticas contempla tres aspectos:

- **Actividades de seguimiento en el laboratorio**.- Representa el 2 % de la nota final y pretende evaluar el grado de implicación del alumno en el desarrollo del proyecto.
- **Evaluación del trabajo en el laboratorio**.- Representa el 3 % de la nota final y pretende evaluar el trabajo continuo en el laboratorio. Esta evaluación se realizará mediante los correspondientes entregables asociados con cada una de las tres partes de las que se compone el proyecto.
- **Evaluación individual del proyecto**.- Representa el 30 % de la nota final. En primer lugar, el compilador del proyectos será APO si:
  - detecta todos los errores léxico, sintácticos (Parte I) y semánticos (Parte II) que aparezcan en los programas de prueba;
  - genera el código intermedio que funcione correctamente para todos los programas de prueba correctos (Parte III).

  En segundo lugar, en el mismo día del examen de teoría del 2º parcial que será el **16 de enero de 2018** (y el 26 de enero de 2018, para una posible recuperación), se realizará un examen práctico individual en el laboratorio. En dicho examen, el alumno deberá demostrar sus conocimientos modificando ligeramente su proyecto para resolver un pequeño problema práctico planteado.

[1] Ver el manual de usuario de los laboratorios docentes del DSIC que se puede encontrar en `/asigDSIC/ETSINF/lppl/doc`

Parte I
Analizador Léxico-Sintáctico

Para la realización de esta parte del proyecto se cuenta con la experiencia adquirida en la resolución de los ejercicios de los seminarios S1: "*Introducción al* FLEX" y S2: "*Introducción al* BISON". En realidad, esta parte puede considerarse una extensión de los ejercicios propuestos en ambos seminarios.

Para facilitar el trabajo, se proporciona el siguiente material auxiliar:

- `Makefile` Un fichero de ejemplo para realizar correctamente la tarea de compilación, carga y edición de enlaces de las distintas partes del proyecto.
- `header.h` En el directorio `include` se ha dejado un ejemplo de un posible fichero de cabeceras donde situar las definiciones de constantes y variables globales de `MenosC`. Obviamente, este fichero deberá modificarse por los alumnos para adaptarlo al desarrollo de su propio proyecto.
- `principal.c` En el directorio `src` se ha dejado un ejemplo de un posible fichero con un programa principal y un tratamiento de errores simple.
- `Programas de prueba` En el directorio `tmp` se han dejado un conjunto de programas de prueba [$a\{0,1,2,3,4\}.c$] para comprobar el funcionamiento de esta parte del compilador.

2. Especificación Léxica de `MenosC`

Para la implementación del Analizador Léxico (AL) para `MenosC` se usará la herramienta FLEX. Las restricciones léxicas que se definen para `MenosC` son las siguientes:

- Los identificadores son cadenas formadas por letras (incluyendo "_") y dígitos, que comienzan siempre por una letra. Se deben distinguir entre mayúsculas y las minúsculas.
- Las palabras reservadas se deben escribir en minúscula. La lista de palabras reservadas puede deducirse fácilmente de la gramática del lenguaje que se define en la Figura 3.
- Aunque puedan aparecer constantes enteras y reales en el programa fuente, todas las constantes numéricas deben considerarse enteras.
- La constante numérica (`cte`) se considera sin signo. El signo + (ó −) debe tratarse como un símbolo léxico independiente.
- Los delimitadores se componen de blancos, retornos de linea y tabuladores. Los delimitadores deben ignorarse, excepto cuando deban separar identificadores, constantes numéricas o palabras reservadas.
- Los comentarios deben ir precedidos por la doble barra (`//`) y terminar con el fin de la linea. Los comentarios pueden aparecer en cualquier lugar donde pueda aparecer un espacio en blanco y solo pueden incluir una linea. Los comentarios no se pueden anidar.

3. Especificación Sintáctica de `MenosC`

Para la implementación del Analizador Sintáctico (AS) de `MenosC` se usará la herramienta BISON. La especificación sintáctica para `MenosC` se define en la Figura 3. Como se puede observar, un programa `MenosC` se compone de una secuencia de sentencias entre llaves, bien sean declaraciones de variables o instrucciones, en cualquier orden.

En la gramática, los símbolos terminales son: separadores; operadores; palabras reservadas (en negrita en la gramática); el símbolo **cte** que representa una constante numérica entera sin signo; y el símbolo **id** que representa un identificador.

programa	→ { secuenciaSentencias }
secuenciaSentencias	→ sentencia | secuenciaSentencias sentencia
sentencia	→ declaracion | instruccion
declaracion	→ tipoSimple **id** ; | tipoSimple **id** [**cte**] ;
tipoSimple	→ **int** | **bool**
instruccion	→ { listaInstrucciones } | instruccionEntradaSalida | instruccionExpresion | instruccionSeleccion | instruccionIteracion
listaInstrucciones	→ listaInstrucciones instruccion | ϵ
instruccionExpresion	→ expresion ; | ;
instruccionEntradaSalida	→ **read** (**id**) ; | **print** (expresion) ;
instruccionSeleccion	→ **if** (expresion) instruccion restoIf
restoIf	→ **elseif** (expresion) instruccion restoIf | **else** instruccion
instruccionIteracion	→ **while** (expresion) instruccion | **do** instruccion **while** (expresion)
expresion	→ expresionLogica | **id** operadorAsignacion expresion | **id** [expresion] operadorAsignacion expresion
expresionLogica	→ expresionIgualdad | expresionLogica operadorLogico expresionIgualdad
expresionIgualdad	→ expresionRelacional | expresionIgualdad operadorIgualdad expresionRelacional
expresionRelacional	→ expresionAditiva | expresionRelacional operadorRelacional expresionAditiva
expresionAditiva	→ expresionMultiplicativa | expresionAditiva operadorAditivo expresionMultiplicativa
expresionMultiplicativa	→ expresionUnaria | expresionMultiplicativa operadorMultiplicatico expresionUnaria
expresionUnaria	→ expresionSufija | operadorUnario expresionUnaria | operadorIncremento **id**
expresionSufija	→ (expresion) | **id** operadorIncremento | **id** [expresion] | **id** | **cte** | **true** | **false**
operadorAsignacion	→ `=` | `+=` | `-=` | `*=` | `/=`
operadorLogico	→ `&&` | `||`
operadorIgualdad	→ `==` | `!=`
operadorRelacional	→ `>` | `<` | `>=` | `<=`
operadorAditivo	→ `+` | `-`
operadorMultiplicativo	→ `*` | `/` | `%`
operadorUnario	→ `+` | `-` | `!`
operadorIncremento	→ `++` | `--`

Figura 1: Especificación sintáctica del lenguaje `MenosC`.18

Parte II
Analizador Semántico

El objetivo de esta segunda parte del proyecto es la implementación, usando BISON, de las restricciones semánticas en general y las comprobaciones de tipos en particular para el lenguaje `MenosC` que se comenzó a desarrollar en la primera fase del proyecto. Además, en esta parte también se deberá realizar la manipulación de la información de los objetos del programa en la Tabla de Símbolos (TDS) y la gestión de memoria estática.

Para facilitar la tarea de codificación se proporciona el siguiente material auxiliar:

- `Makefile`. Una nueva versión que incluye la gestión de una nueva librería.
- `principal.c`, en el directorio `src`. Una versión actualizada para permitir la opción de visualizar o no, la TDS.
- `libtds`. Librería con las operaciones para la manipulación de la TDS.

  En los directorios `include` y `lib` se sitúan respectivamente el fichero con la cabecera, `libtds.h`, y el objeto, `libtds.a`, de la librería.
- `Programas de prueba`. En el directorio `tmp` se encuentra un conjunto de programas de prueba, [$b\{0,1,2,3,4\}.c$], con y sin errores semánticos. Vuestro compilador deberá detectar todos los errores presentes en estos programas de prueba.

4. Especificación semántica

Las restricciones semánticas que se definen para `MenosC` son las siguientes:

- En el compilador solo trabaja con constantes enteras. Si el analizador léxico encuentra una constante real en el programa se debe devolver su valor entero truncado.
- Todas las variables deben declararse antes de ser utilizadas.
- La talla de los tipos simples, *entero* y *lógico*, debe definirse, por medio de la constante TALLA_TIPO_SIMPLE= 1, en el fichero `header.h` del directorio `include`.
- El tipo lógico `bol` se representa numéricamente como un entero: con el valor 0, para el caso `falso`, y 1, para el caso `verdad`.
- No existe conversión de tipos entre `int` y `bol`.
- El operador módulo, " %", realiza el resto de una división entera; por tanto, los dos argumentos deben ser enteros.
- Los índices de los vectores van de `0` a `cte-1`, siendo `cte` el número de elementos definido en su declaración. El número de elementos de un vector debe ser un entero positivo.

- No es necesario comprobar los índices de los vectores en tiempo de ejecución.
- Las expresiones de las instrucciones `if-elseif-else`, `while` y `do-while` deben ser de tipo lógico.
- En cualquier otro caso, las restricciones semánticas por defecto serán las propias del lenguaje ANSI *C*.

5. Gestión de la TDS

En esta sección se presenta la estructura de la *Tabla de Símbolos* que se va a utilizar en la práctica junto con las funciones para su manipulación. Todo esto esta recogido en la librería `libtds` que describiremos a continuación.

5.1. Estructura de la Tabla de Símbolos (TDS)

En el fichero `libtds.h`, del directorio `include`, aparecen las definiciones de las constantes simbólicas, variables globales, estructuras usadas y cabeceras de funciones que serán de utilidad al implementar las acciones semánticas para manipular la TDS. A modo ilustrativo podemos destacar:

- **Constantes simbólicas**, definidas para representar los tipos de los objetos del lenguaje que se utilizan en la librería:

```
/************************ Constantes para los tipos en la Tabla de Símbolos */
#define T_VACIO       0
#define T_ENTERO      1
#define T_LOGICO      2
#define T_ARRAY       3
#define T_ERROR       4
```

- **Estructuras básicas**, que contienen la información de la TDS para los objetos simples y vectores. Posteriormente se verá que alguna función de consulta a la TDS devuelve estas estructuras.

```
typedef struct simb /****************************** Estructura para la TDS */
{ int   tipo;           /* Tipo del objeto                                  */
  int   desp;           /* Desplazamiento relativo en el segmento variables */
  int   ref;            /* Campo de referencia de usos múltiples            */
} SIMB;

typedef struct dim  /*********** Estructura para la información de un vector */
{ int   telem;                                  /* Tipo de los elementos */
  int   nelem;                                  /* Número de elementos   */
} DIM;
```

- **Variables globales**, de uso en todo el compilador:

```
int dvar;            /* Desplazamiento relativo en el Segmento de Variables  */
```

5.2. Funciones de manipulación de la TDS

En la Figura 2 se presenta el listado de las funciones que deben emplearse para acceder a la TDS.

```
int insertarTDS (char *nom, int tipo, int desp, int ref) ;
/* Inserta en la TDS toda la información asociada con un objeto definido por
   el usuario: nombre ``nom''; tipo ``tipo''; desplazamiento relativo en el
   segmento de variables ``desp'' y referencia ``ref'' a una posible subtabla
   de vectores. Donde, ``ref = -1'', para los objetos de tipo simple. Si el
   objeto ya existe devuelve el valor ``FALSE = 0'' (``TRUE = 1'' en caso
   contrario).  */

int insertaTDArray (int telem, int nelem) ;
/* Inserta en la Tabla de Arrays la información de un array cuyos elementos
   son de tipo ``telem'' y el número de elementos es ``nelem''. Devuelve su
   referencia en la Tabla de Arrays.  */

SIMB obtenerTDS (char *nom) ;
/* Obtiene toda la información asociada con un objeto de nombre ``nom'' y
   la devuelve en una estructura de tipo ``SIMB'' (ver libtds.h). Si
   el objeto no está declarado, en el campo ``tipo'' devuelve ``T_ERROR''.  */

DIM obtenerInfoArray (int ref) ;
/* Obtiene la información de un array referenciado por ``ref'' en la Tabla
   de Arrays y la devuelve en una estructura de tipo ``DIM'' (ver libtds.h).  */

void mostrarTDS () ;
/* Muestra toda la información de la TDS.  */
```

Figura 2: Perfil de las funciones de manipulación de la TDS.

6. Ejemplos ilustrativos

En esta sección se muestran dos ejemplos sencillos de comprobación de tipos: uno en la fase de declaración de variables (inferencia de tipos) y otro en el de las expresiones (comprobación de tipos).

6.1. Comprobación de tipos en *declaraciones*

Para la declaración de un objeto elemental de tipo array, un posible ejemplo de comprobación de tipos y de gestión estática de memoria podría ser:

```
declaracion | tipoSimple ID_ ACOR_ CTE_ CCOR_ PUNTOCOMA_

            { int numelem = $4; int refe;
              if ($4 <= 0)  {
                yyerror("Talla inapropiada del array");
                numelem = 0;
              }
              refe = insertaTDArray($1, numelem);
              if ( ! insertarTDS($2, T_ARRAY, dvar, refe) )
                yyerror ("Identificador repetido");
              else dvar += numelem * TALLA_TIPO_SIMPLE;
            }
```

6.2. Comprobación de tipos en *expresiones de asignación*

En el caso de una expresión de asignación, donde se espera que los operandos sean de tipo simple, su comprobación de tipos podría ser:

```
expresion
       | ID_  ASIGNACION_  expresion  PUNTOCOMA_

         { SIMB sim = obtenerTDS($1); $$ = T_ERROR;

           if (sim.tipo == T_ERROR) yyerror("Objeto no declarado");
           else if (! ((sim.tipo == $3 == T_ENTERO) ||
                       (sim.tipo == $3 == T_LOGICO)))
             yyerror("Error de tipos en la 'instrucción de asignación'");
           else $$ = sim.tipo;
         }
```

Advertid que para evitar una secuencia de errores redundantes debería modificarse este código para que solo se de un nuevo mensaje de error si el error se produce en esta regla, y no si proviene de errores anteriores a través de $1 o $3.

7. Recomendaciones finales de implementación

7.1. Atributos léxicos

Para trabajar con los atributos de los símbolos del lenguaje, en primer lugar, hay que definir el conjunto de posibles tipos de atributos. Para ello:

1. Especificar la colección completa de los (tipos de) atributos en una declaración `%union` en `Bison`; por ejemplo, para los atributos léxicos podríamos definir:

```
%union {
   char *ident;        /* Nombre del identificador */
   int cent;           /* Valor de la cte numérica entera */
}
```

2. Evaluar los atributos léxicos asociados con los *identificadores* y las *constantes enteras*. Los atributos de los terminales se asignan a la variable `yylval` en las reglas del `Flex` donde se define cada token. Por ejemplo:

```
{numero}        { yylval.cent  = atoi(yytext);   return(CTE_); }
{identificador} { yylval.ident = strdup(yytext); return(ID_);  }
```

   Donde `CTE_` y `ID_` son codificaciones arbitrarias para las *constantes enteras* y los *identificadores* de variables.

7.2. Fichero de cabeceras

Las constantes, estructuras y variables globales que se utilicen en todo el compilador, es conveniente definirlas en vuestro fichero de cabecera `header.h` (y situarlo en el directorio `include`). Algunas sugerencias para añadir a vuestro `header.h` de la Parte-1 podrían ser:

- **Constantes simbólicas**

```
/************************************ Tallas asociadas a los tipos simples */
#define TALLA_TIPO_SIMPLE     1
```

- **Variables Globales**

```
/*********************** Variables externas definidas en Programa Principal */
extern int verTDS;                  /* Flag para saber si mostrar la TDS     */

/***************************** Variables externas definidas en las librerías */
extern int dvar;                /* Desplazamiento en el Segmento de Variables  */
```

Parte III
Generador de Código Intermedio

Teniendo en cuenta el trabajo desarrollado en las dos primeras fases del proyecto, el objetivo de esta tercera parte es dotar al compilador de `MenosC` de la etapa de *Generación de Código*; en realidad, de un código intermedio denominado `Malpas` y para el que existe una máquina virtual que llamaremos `mvm` (Máquina Virtual `Malpas`).

Para facilitar esta tarea, se proporciona el siguiente material auxiliar:

- `Makefile`. Una nueva versión que incluye la gestión de una nueva librería.
- `principal.c`, en el directorio `src`. Una versión actualizada para permitir el volcado del código generado en el proceso de compilación.
- `libgci`. Librería con las operaciones para la generación de código intermedio. Como en el caso de la librería anterior, el fichero de cabeceras, `libgci.h`, se sitúa en el directorio `include` y la propia librería, `libgci.a`, en el directorio `lib`.
- `mvm`. Máquina virtual para la ejecución del código intermedio `Malpas` generado por vuestro compilador y está disponible en un directorio de nombre `bin`.
- `Programas de prueba`. En el directorio `tmp` se han dejado un conjunto de programas de prueba, [$c\{0,1,2,3,4,5,6\}.c$], sin errores. Estos programas, junto con los proporcionados para las comprobaciones sintácticas y semánticas, constituyen los programas de evaluación de la práctica. Para que la práctica pueda ser calificada como APTA será condición necesaria que el compilador genere código intermedio correcto para estos programas. La comprobación de la corrección del código generado se realizará mediante la ejecución del código intermedio en la máquina virtual `mvm`.

8. La máquina virtual `Malpas`

Tal y como se ha comentado, el objetivo de este proyecto es la construcción de un compilador para el lenguaje `MenosC`, que genere código `Malpas` para una máquina virtual `mvm`.

8.1. Inventario de instrucciones `Malpas`

En esta sección se presenta el juego de instrucciones de `Malpas`, agrupadas por categorías. Para cada instrucción se distinguen cuatro partes: *código de operación* (OP); dos *argumentos* (arg1 y arg2) y un *resultado* (res). Además se proporciona una pequeña leyenda con su *significado*. Tanto los argumentos como el resultado pueden ser: *enteros* (I); *posición* (P); *etiquetas* (E) o *nulo* (vacío).

Operaciones aritméticas

OP	arg1	arg2	res	Significado
ESUM	I/P	I/P	P	Suma
EDIF	I/P	I/P	P	Resta
EMULT	I/P	I/P	P	Multiplicación
EDIVI	I/P	I/P	P	División entera
RESTO	I/P	I/P	P	Resto división entera
ESIG	I/P		P	Cambio de signo
EASIG	I/P		P	Asignación

Operaciones de salto

OP	arg1	arg2	res	Significado
GOTOS			E	Salto incondicional a E
EIGUAL	I/P	I/P	E	si arg1=arg2 salto a E
EDIST	I/P	I/P	E	si arg1<>arg2 salto a E
EMEN	I/P	I/P	E	si arg1<arg2 salto a E
EMAY	I/P	I/P	E	si arg1>arg2 salto a E
EMENEQ	I/P	I/P	E	si arg1<=arg2 salto a E
EMAYEQ	I/P	I/P	E	si arg1>=arg2 salto a E

Operaciones con direccionamiento relativo (vectores)

OP	arg1	arg2	res	Significado
EAV	P	I/P	P	Asigna un elemento de un vector a una variable: $res := arg1[arg2]$
EVA	P	I/P	P	Asigna una variable a un elemento de un vector: $arg1[arg2] := res$

Operaciones de entrada/salida

OP	arg1	arg2	arg3	Significado
EREAD			P	Lectura
EWRITE			I/P	Escritura

Operaciones adicionales

OP	arg1	arg2	res	Significado
FIN				Fin del programa

8.2. Arquitectura de la máquina virtual `mvm`

La gestión de memoria de `mvm` es completamente estática; es decir, solo dispondrá de un segmento (estático) de memoria para almacenar todas las variables definidas en el programa (de usuario y temporales). La estructura de memoria para `mvm` se puede observar en la Figura 3, donde `OrSegEst` representa el origen del segmento de variables y será gestionado por `mvm`.

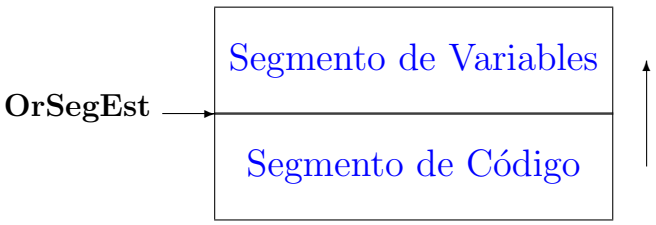

Figura 3: Estructura de memoria para `mvm`.

8.2.1. Acceso a las variables en `mvm`

El acceso a la dirección física de memoria donde se encuentra un objeto es responsabilidad de `mvm` y lo realizará internamente. Para ello es necesario proporcionarle (en el argumento correspondiente de la instrucción 3-direcciones) el *desplazamiento relativo* al segmento. Para calcular esta dirección física, `mvm` internamente sumará el desplazamiento relativo del objeto a la dirección base del segmento de variables estáticas, `OrSegEst`.

9. Generación de código intermedio

Al finalizar la compilación de un programa fuente se debe generar el código objeto asociado. En nuestro caso, este código objeto será la secuencia de código intermedio producido y se deberá volcar a un fichero (texto). Este volcado debe realizarse siempre y cuando no se hayan detectado errores en la compilación. Las instrucciones de código intermedio 3-direcciones tienen el siguiente formato:

```
<Cod_Operación>  <arg1>  <arg2>  <res>
```

Donde para cada uno de los argumentos ($arg1$, $arg2$ y res) se debe indicar: el tipo del argumento (**p** para *posición*, **i** para *entero*, **e** para *etiqueta* o vacío para *nulo*) y su valor.

La generación código se realizará con la ayuda de la función `volcarCodigo` de la librería `libgci`.

9.1. Estructuras de datos y variables globales

En el fichero `libgci.h` se encuentran la definición de las variables globales, las constantes simbólicas, las estructuras principales y las cabeceras de las funciones necesarias para la generación de código tres direcciones.

- **Constantes simbólicas**, definidas para representar el código de las instrucciones tres-direcciones y el tipo de sus argumentos.

```
/********* Constantes para el tipo de los argumentos de las instrucciones 3D */
#define ARG_ENTERO    0
#define ARG_POSICION  1
#define ARG_ETIQUETA  2
#define ARG_NULO      3
/********************************** Instrucciones del Código Tres Direcciones */
#define ESUM          0
#define EDIF          1
#define EMULT         2
.......................
#define FIN          18
```

- **Variables globales**, de uso en todo el compilador:

```
/************************** Variables globales de uso en todo el compilador */
int si;                        /* Desplazamiento en el Segmento de Código     */
```

Como se puede apreciar, las librerías definen y manejan dos variables globales para gestionar el desplazamiento relativo de los objetos: `dvar` en el segmento de datos (definida en `libgts.h`) y **si** en el segmento de código (definida en `libgci.h`).

- **Estructuras básicas**, que contienen la información necesaria para la generación de código. Estas estructuras son necesarias para las funciones que se verán a continuación.

```
/****************************** Estructura para los argumentos del código 3D */
typedef struct tipo_arg {
  int tipo;        /* Tipo  del argumento: entero, posición, etiqueta o nulo */
  int val;         /* Valor del argumento: entero, posición, etiqueta o nulo */
} TIPO_ARG;
```

TIPO_ARG es un tipo que representa los argumentos de las instrucciones 3-direcciones. Por ejemplo, para almacenar un argumento de tipo entero con valor 5, basta con realizar las asignaciones `arg1.tipo`=ARG_ENTERO; `arg1.val`= 5. No obstante, no recomendamos realizar este tipo de asignaciones ya que se dispone de *funciones* específicas (presentadas en la siguiente sección) que permiten hacerlo de una manera mucho más sencilla: `crArgEnt`(), `crArgEtq`(), `crArgPos`() y `crArgNul`().

9.2. Funciones de ayuda a la GCI

Las funciones que se definen en la librería `libgci.h` son:

```
/***************** Funciones para facilitar la tarea de generacion de codigo */
void emite (int cop, TIPO_ARG arg1, TIPO_ARG arg2, TIPO_ARG res);
/* Crea una instrucción tres direcciones con el código de operación, "cop", y
   los argumentos "arg1", "arg2" y "res", y la pone en la siguiente posición
   libre (indicada por "si") del Segmento de Código. A continuación,
   incrementa "si".                                                          */
int creaVarTemp ();
/*  Crea una variable temporal de tipo simple (TALLA_TIPO_SIMPLE = 1), en el
    segmento de variables (indicado por "dvar") y devuelve su desplazamiento
    relativo. A continuación, incrementa "dvar".                             */
void volcarCodigo (char *nom) ;
/* Vuelca (en modo texto) el código generado en un fichero cuyo nombre es el
   del fichero de entrada con la extensión ".c3d".                           */
/*************** Funciones para crear los argumentos de las instrucciones 3D */
TIPO_ARG crArgNul () ;
/* Crea el argumento de una instrucción tres direcciones de tipo nulo.       */
TIPO_ARG crArgEnt (int valor) ;
/* Crea el argumento de una instrucción tres direcciones de tipo entero
   con la información de la constante entera dada en "valor".                */
TIPO_ARG crArgEtq (int valor) ;
/*  Crea el argumento de una instrucción tres direcciones de tipo etiqueta
    con la información de la dirección dada en "valor".                      */
TIPO_ARG crArgPos (int valor) ;
/*  Crea el argumento de una instrucción tres direcciones de tipo posición
    con la información del desplazamiento relativo dada en "valor".          */
/********************************* Funciones para la manipulación de las LANS */
int creaLans (int d);
/* Crea una lista de argumentos no satisfechos para una instrucción
   incompleta cuya dirección es "d" y devuelve su referencia.                */
int fusionaLans (int x, int y);
/* Fusiona dos listas de argumentos no satisfechos cuyas referencias
   son "x" e "y" y devuelve la referencia de la lista fusionada.             */
void completaLans (int x, TIPO_ARG arg);
/* Completa con el argumento "arg" el campo "res" de todas las instrucciones
   incompletas de la lista "x".                                              */
```

9.3. Recomendación final de implementación

El fichero de cabeceras `header.h` debería incluir también:

```
/************************** Variables externas definidas en las librerías ***/
extern int si;          /* Desplazamiento relativo en el Segmento de Código */
```

10. Ejemplos ilustrativos

10.1. Generación de código intermedio

A continuación se muestra un ejemplo sencillo de generación de código intermedio para la `expresionAditiva`. Recordar que para evitar una secuencia de errores redundantes

debería modificarse la comprobación de tipos para que solo se de un nuevo mensaje de error si el error se produce en esta regla, y no si proviene de errores anteriores a través de $1 o $3.

```
operadorAditivo
       : MAS_          { $$ = ESUM; }
       | MENOS_        { $$ = EDIF; } ;
expresionAditiva
       : expresionMultiplicativa        { $$ = $1; }
       | expresionAditiva operadorAditivo expresionMultiplicativa
         {
           $$.tipo = T_ERROR;
           if ($1.tipo == $3.tipo == T_ENTERO) $$.tipo = T_ENTERO;
           else yyerror("Error de tipos en la 'expresión aditiva'");
           $$.pos = creaVarTemp();
           /***************** Expresión a partir de un operador aritmético */
           emite($2, crArgPos($1.pos), crArgPos($3.pos), crArgPos($$.pos));
         } ;
```

10.2. Programa en código intermedio

En esta sección se presenta un ejemplo de código generado para un pequeño programa que calcula el factorial de un número. Se trata solo de un ejemplo de cómo se podría generar el código intermedio, y por lo tanto, distintos compiladores podrán generar código diferente pero igualmente válido.

```
// Calcula el factorial de un número > 0 y < 13
{ int n; int fac; int i; bool f;

  f = true; fac = 1;
  while ( f ) {
    read(n);
    if ((n > 0) && (n < 13)) {
      i = 2;
      while (i <= n) { fac = fac * i; i++; }
      print(fac); f = false;
    }
    else {}
  }
}
```

Y el código tres direcciones será:

0	EASIG	i: 1	,		, p: 4
1	EASIG	p: 4	,		, p: 3
2	EASIG	p: 3	,		, p: 5
3	EASIG	i: 1	,		, p: 6
4	EASIG	p: 6	,		, p: 1
5	EASIG	p: 1	,		, p: 7
6	EASIG	p: 3	,		, p: 8
7	EIGUAL	p: 8	, i: 0	, e: 45
8	EREAD		,		, p: 0
9	EASIG	p: 0	,		, p: 9
10	EASIG	i: 0	,		, p: 10
11	EASIG	i: 1	,		, p: 11
12	EMAY	p: 9	, p: 10	, e: 14
13	EASIG	i: 0	,		, p: 11
14	EASIG	p: 0	,		, p: 12
15	EASIG	i: 13	,		, p: 13
16	EASIG	i: 1	,		, p: 14
17	EMEN	p: 12	, p: 13	, e: 19
18	EASIG	i: 0	,		, p: 14
19	EMULT	p: 11	, p: 14	, p: 15
20	EIGUAL	p: 15	, i: 0	, e: 44
21	EASIG	i: 2	,		, p: 16
22	EASIG	p: 16	,		, p: 2

23	EASIG	p: 2	,		, p: 17
24	EASIG	p: 2	,		, p: 18
25	EASIG	p: 0	,		, p: 19
26	EASIG	i: 1	,		, p: 20
27	EMENEQ	p: 18	, p: 19	, e: 29
28	EASIG	i: 0	,		, p: 20
29	EIGUAL	p: 20	, i: 0	, e: 38
30	EASIG	p: 1	,		, p: 21
31	EASIG	p: 2	,		, p: 22
32	EMULT	p: 21	, p: 22	, p: 23
33	EASIG	p: 23	,		, p: 1
34	EASIG	p: 1	,		, p: 24
35	EASIG	p: 2	,		, p: 25
36	ESUM	p: 2	, i: 1	, p: 2
37	GOTOS		,		, e: 24
38	EASIG	p: 1	,		, p: 26
39	EWRITE		,		, p: 26
40	EASIG	i: 0	,		, p: 27
41	EASIG	p: 27	,		, p: 3
42	EASIG	p: 3	,		, p: 28
43	GOTOS		,		, e: 44
44	GOTOS		,		, e: 6
45	FIN		,		,