

Lenguajes de Programación y Procesadores de Lenguajes

Construcción de un compilador para **MenosC** (2020-2021)

Partes I, II y III

14 de septiembre de 2020

Índice

1. Introducción y justificación del proyecto	1
1.1. Presentación y objetivos	1
1.2. Material de prácticas	2
1.3. Evaluación	2
I Analizador Léxico-Sintáctico	3
2. Especificación Léxica de MenosC	3
3. Especificación Sintáctica de MenosC	4
4. Recomendaciones adicionales	4
II Analizador Semántico	6
5. Especificación semántica	6
6. Gestión de la TdS	7
6.1. Estructura de la Tabla de Símbolos (TdS)	7
6.2. Funciones de manipulación de la TdS	8
7. Ejemplos ilustrativos	8
7.1. Comprobación de tipos en <i>declaraciones</i>	8
7.2. Comprobación de tipos en las <i>expresiones</i>	10
8. Recomendaciones de implementación	10
8.1. Atributos léxicos	10
8.2. Fichero de cabeceras	11
III Generador de Código Intermedio	12
9. La máquina virtual Malpas	12
9.1. Inventario de instrucciones Malpas	12
9.2. Arquitectura de la máquina virtual mvm	14
9.2.1. Estructura de los RA	15
9.2.2. Acceso a las variables en mvm	15
10. Generación de código intermedio	15
10.1. Estructuras de datos y variables globales	15
10.2. Funciones de ayuda a la GCI	16
10.3. Recomendación final de implementación	17
11. Ejemplos ilustrativos	17
11.1. Generación de código intermedio	17
11.2. Programa en código intermedio	18

1. Introducción y justificación del proyecto

Las practicas de la asignatura de *Lenguajes de Programación y Procesadores de Lenguajes* 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 adquiriera una visión más realista del funcionamiento de un compilador.
2. Aumentar la motivación del alumno, ya que se le plantean auténticos retos reales: construcción de un compilador.
3. Facilitar la comprensión de algunos conceptos que difícilmente se entenderían solo en las sesiones teóricas.
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.
5. Desarrollar capacidades transversales como el trabajo en equipo, las relaciones interpersonales, la comunicación, toma de decisiones y manejo del tiempo.

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 estos documentos, consulte los manuales en línea de FLEX y BISON que se encuentran disponibles en ([/asigDSIC/ETSINF/lppl/doc](#)), o 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 6 % de la nota final y pretende evaluar el trabajo continuo en el laboratorio y el grado de implicación del alumno en el desarrollo del proyecto. 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 APTO si,
 - detecta todos los errores léxicos, sintácticos (Parte I) y semánticos (Parte II) que aparezcan en los programas de prueba, y
 - 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 **15 de enero de 2021** (y el 27 de enero de 2021, 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.

¹Ver 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, en (`/asigDSIC/ETSINF/lpp1/pry/P1/`), 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`. 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`. Un ejemplo de un posible fichero con un programa principal y un tratamiento de errores simple.
- **Programas de prueba**, en el directorio `tmp`. 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**². Las restricciones léxicas que se definen para **MenosC** son las siguientes:

- Los nombres de variables pueden contener letras (incluyendo “_”) y dígitos, y deben comenzar siempre por una letra. **MenosC** distingue entre mayúsculas y minúsculas. La costumbre es que las variables van en minúscula y las constantes en mayúscula.
- Las palabras reservadas deben escribirse en minúscula. La lista de palabras reservadas puede deducirse fácilmente de la gramática del lenguaje que se define en la Figura 1.
- En un programa fuente puedan aparecer constantes enteras y reales; por ejemplo:
`28 28. .55 28.55`
- La constante numérica (**cte**) se considera sin signo. El signo + (ó –) debe tratarse como un símbolo léxico independiente.

²Su manual puede encontrarse en `/asigDSIC/ETSINF/lpp1/doc/`

- Los comentarios deben ir precedidos por la doble barra (//) y terminar con el fin de la línea. Los comentarios pueden aparecer en cualquier lugar donde pueda aparecer un espacio en blanco y solo pueden incluir una línea. Los comentarios no se pueden anidar.
- Los delimitadores se componen de blancos, retornos de línea y tabuladores. Los delimitadores deben ignorarse, excepto cuando deban separar identificadores o palabras reservadas.

3. Especificación Sintáctica de MenosC

Para la implementación del Analizador Sintáctico (AS) de **MenosC** se usará la herramienta BISON³. La especificación sintáctica para **MenosC** se define en la Figura 1. Como se puede observar, un programa **MenosC** se compone de una secuencia de declaraciones, bien sean variables o funciones, 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.

4. Recomendaciones adicionales

Además de completar el *Analizador Léxico-Sintáctico* para **MenosC**, en esta primera parte se deben llevar a cabo las siguientes recomendaciones adicionales:

1. Implementar la opción *verbosidad*. Analizando el programa `principal.c` se puede observar que la ejecución del compilador tiene un parámetro opcional: `-v`. Este parámetro indica que el usuario desea una información detallada del proceso de compilación. Esto se consigue mediante una variable global denominada *verbosidad*
2. Con la opción (*verbosidad* = `true`) se debe reproducir el programa fuente analizado, añadiendo además el número de línea.
3. La gramática de **MenosC** no debe modificarse en ningún caso.

³Su manual puede encontrarse en `/asigDSIC/ETSINF/lpp1/doc/`

programa	→ listaDeclaraciones
listaDeclaraciones	→ declaracion listaDeclaraciones declaracion
declaracion	→ declaracionVariable declaracionFuncion
declaracionVariable	→ tipoSimple id ; tipoSimple id [cte] ;
tipoSimple	→ int bool
declaracionFuncion	→ cabeceraFuncion bloque
cabeceraFuncion	→ tipoSimple id (parametrosFormales)
parametrosFormales	→ ϵ listaParametrosFormales
listaParametrosFormales	→ tipoSimple id tipoSimple id , listaParametrosFormales
bloque	→ { declaracionVariableLocal listaInstrucciones return expresion ; }
declaracionVariableLocal	→ ϵ declaracionVariableLocal declaracionVariable
listaInstrucciones	→ ϵ listaInstrucciones instruccion
instruccion	→ { listaInstrucciones } instruccionAsignacion instruccionSeleccion instruccionEntradaSalida instruccionIteracion
instruccionAsignacion	→ id = expresion ; id [expresion] = expresion ;
instruccionEntradaSalida	→ read (id) ; print (expresion) ;
instruccionSeleccion	→ if (expresion) instruccion else instruccion
instruccionIteracion	→ for (expresionOpcional ; expresion ; expresionOpcional) instruccion
expresionOpcional	→ ϵ expresion id = expresion
expresion	→ expresionIgualdad expresion operadorLogico expresionIgualdad
expresionIgualdad	→ expresionRelacional expresionIgualdad operadorIgualdad expresionRelacional
expresionRelacional	→ expresionAditiva expresionRelacional operadorRelacional expresionAditiva
expresionAditiva	→ expresionMultiplicativa expresionAditiva operadorAditivo expresionMultiplicativa
expresionMultiplicativa	→ expresionUnaria expresionMultiplicativa operadorMultiplicativo expresionUnaria
expresionUnaria	→ expresionSufija operadorUnario expresionUnaria operadorIncremento id
expresionSufija	→ (expresion) id operadorIncremento id [expresion] id (parametrosActuales) id constante
parametrosActuales	→ ϵ listaParametrosActuales
listaParametrosActuales	→ expresion expresion , listaParametrosActuales
constante	→ cte true false
operadorLogico	→ &&
operadorIgualdad	→ == !=
operadorRelacional	→ > < >= <=
operadorAditivo	→ + -
operadorMultiplicativo	→ * /
operadorUnario	→ + - !
operadorIncremento	→ ++ --

Figura 1: Especificación sintáctica del lenguaje **MenosC**

Parte II

Analizador Semántico

El objetivo de la segunda parte del proyecto es la implementación 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 parte 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, en (`/asigDSIC/ETSINF/lpp1/pry/P2/`), 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 las cabeceras, `libtds.h`, y el objeto, `libtds.a`, de la librería.

- **Programas de prueba**, en el directorio `tmp`. Un conjunto de programas de prueba, `[b{0,1,2,3,4,5}.c]`, con y sin errores semánticos. El compilador deberá detectar todos los errores presentes en estos programas de prueba.

5. Especificación semántica

Para la implementación del Analizador Semántico para **MenosC** se usará la herramienta BISON⁴. Las restricciones semánticas que se definen para **MenosC** son las siguientes:

- Todas las variables y funciones deben declararse antes de ser utilizadas.
- Debe haber una función, y solo una, con el nombre `main`. Esta función determinará el inicio y el fin en la ejecución el programa.
- La información de los parámetros se situará, en la Tabla de Símbolos, en orden inverso a su declaración.
- El paso de parámetros se hace siempre por valor.
- Se admite la recursividad en las funciones.
- El compilador solo trabaja con constantes enteras. Si el analizador léxico encuentra una constante real en el programa debe devolver su valor entero truncado.

⁴Su manual puede encontrarse en `/asigDSIC/ETSINF/lpp1/doc`

- 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`.
- 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`.
- 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.
- Los operadores de incremento y decremento (infijos y postfijos) solo se pueden aplicar a variables (p.ej. `(i+j)++` es ilegal).
- La instrucción `for` es similar a la del C, excepto que las expresiones opcionales deben ser asignaciones o expresiones, pudiendo no aparecer explícitamente. La expresión, como en C, debe ser de tipo lógico y debe aparecer explícitamente.
- La expresión de la instrucción `if-else` debe ser de tipo lógico.
- En cualquier otro caso, las restricciones semánticas por defecto serán las propias del lenguaje ANSI C.

6. 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 está recogido en la librería `libtds` que describiremos a continuación.

6.1. 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 TdS */
#define T_VACIO          0
#define T_ENTERO         1
#define T_LOGICO         2
#define T_ARRAY          3
#define T_ERROR          4
/***** Constantes para las categorías en la TdS */
#define NULO             0
#define VARIABLE         1
#define FUNCION          2
#define PARAMETRO        3

```

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

```
int dvar;           /* Desplazamiento relativo en el Segmento de Variables */
int niv;           /* Nivel de anidamiento: "global" o "local" */
```

- **Estructura básica**, que contienen la información de la TdS para los objetos simples, vectores y funciones. En la sección 6.2 veremos algunas funciones de consulta a la TdS que devuelven estas estructuras.

```
typedef struct simb { /****** Estructura para la TdS */
    int t;           /* Tipo del objeto */
    int n;           /* Nivel "global" o "local" */
    int d;           /* Desplazamiento relativo */
    int ref;         /* Campo de referencia de usos multiples */
} SIMB;
typedef struct dim { /****** Estructura para los vectores */
    int telem;       /* Tipo de los elementos */
    int nelem;       /* Número de elementos */
} DIM;
typedef struct inf { /****** Estructura para las funciones */
    char *nom;       /* Nombre de la función */
    int tipo;        /* Tipo del rango de la función */
    int tsp;         /* Talla del segmento de parámetros */
} INF;
```

6.2. 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.

7. Ejemplos ilustrativos

7.1. 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_ AC_ CTE_ CC_ PCOMA_

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

```

void cargaContexto (int n) ;
/* Crea el contexto necesario para los objetos globales y para los objetos
   locales a las funciones */
void descargaContexto (int n) ;
/* Libera en la TdB y la TdS el contexto asociado con la función. */

int insTdS (char *nom, int cat, int tipo, int n, int desp, int ref) ;
/* Inserta en la TdS toda la información asociada con una variable de nombre,
   "nom"; categoría, "cat"; tipo, "tipo"; nivel del bloque, "n"; desplaza-
   miento relativo, "desp"; y referencia, "ref", a posibles subtablas de
   vectores o dominios, siendo (-1) si es de tipo simple. Si la variable ya
   existe devuelve el valor "FALSE=0" ("TRUE=1" en caso contrario). */
SIMB obtTdS (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, devuelve "T_ERROR" en el campo "tipo". */

int insTdA (int telem, int nelem) ;
/* Inserta en la Tabla de array la información de un array con elementos de
   tipo, "telem"; y número de elementos, "nelem". Devuelve su referencia en
   la Tabla de Arrays. */
DIM obtTdA (int ref) ;
/* Devuelve toda la información asociada con un array referenciado por "ref"
   en la Tabla de Arrays. En caso de error devuelve "T_ERROR" en el campo
   "telem". */

int insTdD (int refe, int tipo) ;
/* Para un dominio existente referenciado por "refe", inserta en la Tabla
   de Dominios la información del "tipo" del parámetro. Si "refe = -1" entonces
   crea una nueva entrada en la tabla de dominios para el tipo de este
   parámetro y devuelve su referencia. Si la función no tiene parámetros,
   debe crearse un dominio vacío con: "refe = -1" y "tipo = T_VACIO". */
INF obtTdD (int refe) ;
/* Si "refe<0" entonces devuelve la información de la función actual, y si
   "refe>=0", devuelve la información de una función ya compilada con
   referencia "refe". La información es: el nombre y el tipo del rango de la
   función y la talla del segmento de parámetros. Si "refe" no se corresponde
   con una función ya compilada, devuelve "T_ERROR" en el campo "tipo". */
int cmpDom (int refx, int refy) ;
/* Si los dominios referenciados por "refx" y "refy" no coinciden devuelve
   "FALSE=0" ("TRUE=1" si son iguales). */

void mostrarTdS () ;
/* Muestra toda la información de la TdS para objetos globales y locales.
   Se recomienda hacerlo (si "verTdS = true") al finalizar la compilación
   de la función, justo antes de ("descargarContexto"). */

```

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

7.2. Comprobación de tipos en las *expresiones*

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

```
instruccionAsignacion
| ID_ ASIG_ expresion PCOMA_

{ SIMB sim = obtTdS($1);

  if (sim.t == T_ERROR) yyerror("Objeto no declarado");
  else if (!(sim.t == $3.t == T_ENTERO) ||
            (sim.t == $3.t == T_LOGICO))
    yyerror("Error de tipos en la 'instrucción de asignación'");
}
```

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.

8. Recomendaciones de implementación

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

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

8.2. Fichero de cabeceras

Las constantes, estructuras y variables globales que se utilicen en todo el compilador, es conveniente definir las 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

```
#define TALLA_TIPO_SIMPLE 1      /* Talla asociada a los tipos simples */
#define TALLA_SEGENLACES 2      /* Talla del segmento de Enlaces de Control */
```

- 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 */
extern int niv;                       /* Nivel de anidamiento "global" o "local" */
```

Parte III

Generador de Código Intermedio

Teniendo en cuenta el trabajo desarrollado en las dos primeras partes del proyecto, el objetivo de esta tercera parte es agregar la etapa de *Generación de Código* a vuestro compilador de **MenosC**. En realidad, este código es un código intermedio que denominaremos **Malpas** y para el que existe una máquina virtual que llamaremos **mvm** (Máquina Virtual Malpas) que permitirá la ejecución de dicho código.

Para facilitar esta tarea de codificación, en ([/asigDSIC/ETSINF/lppl/pry/P3/](#)), 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**, en el directorio [bin](#). Una Máquina virtual para la ejecución del código intermedio Malpas generado por vuestro compilador.
- **Programas de prueba**, en el directorio [tmp](#). Un conjunto de programas de prueba, [*c{0,1,2,3,4,5,6,7}.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 de dicho código intermedio en la máquina virtual **mvm**.

9. 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**.

9.1. 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
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 <i>res</i>
EIGUAL	I/P	I/P	E	si $arg1 == arg2$ salto a <i>res</i>
EDIST	I/P	I/P	E	si $arg1 <> arg2$ salto a <i>res</i>
EMEN	I/P	I/P	E	si $arg1 < arg2$ salto a <i>res</i>
EMAY	I/P	I/P	E	si $arg1 > arg2$ salto a <i>res</i>
EMENEQ	I/P	I/P	E	si $arg1 \leq arg2$ salto a <i>res</i>
EMAYEQ	I/P	I/P	E	si $arg1 \geq arg2$ salto a <i>res</i>

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 de llamada

OP	arg1	arg2	res	Significado
FIN				Fin del programa
RET				Desapila la dirección de retorno y transfiere el control a dicha dirección
CALL			E	Apila la dirección de retorno y transfiere el control a <i>res</i>

Operaciones de manejo de pila de RA

OP	arg1	arg2	res	Significado
EPUSH			I/P	Apila <i>res</i> en la cima de la pila
EPOP			P	Desapila la cima de la pila y la deposita en <i>res</i>
PUSHFP				Apila el FP [†] en la cima de la pila
FPPOP				Desapila la cima y la deposita en el FP
FPTOP				El FP apunta a la misma posición que la cima de la pila
TOPFP				La cima de la pila apunta a la misma posición que el FP
INCTOP			I	Incrementa la cima de la pila en <i>res</i> posiciones
DECTOP			I	Decrementa la cima de la pila en <i>res</i> posiciones

[†] FP \equiv *frame pointer*, puntero al segmento de variables locales y temporales del Registro de Activación actual.

9.2. Arquitectura de la máquina virtual mvm

La gestión memoria de **mvm** se gestiona como un pila de Registros de Activación (RA), o “*frames*”, asociados con cada una de las funciones definidas por el usuario. En el RA se gestiona la memoria para las variables locales y parámetros de la función correspondiente. A efectos de simplificar y homogeneizar la gestión de la memoria, las variables globales (o externas) se consideran asociadas a un RA de nivel cero (o inicial) (ver la Figura 3). Dado que **Malpas** no permite objetos dinámicos, la gestión de memoria de **mvm** se reducirá pues a la manipulación de esta pila de RA.

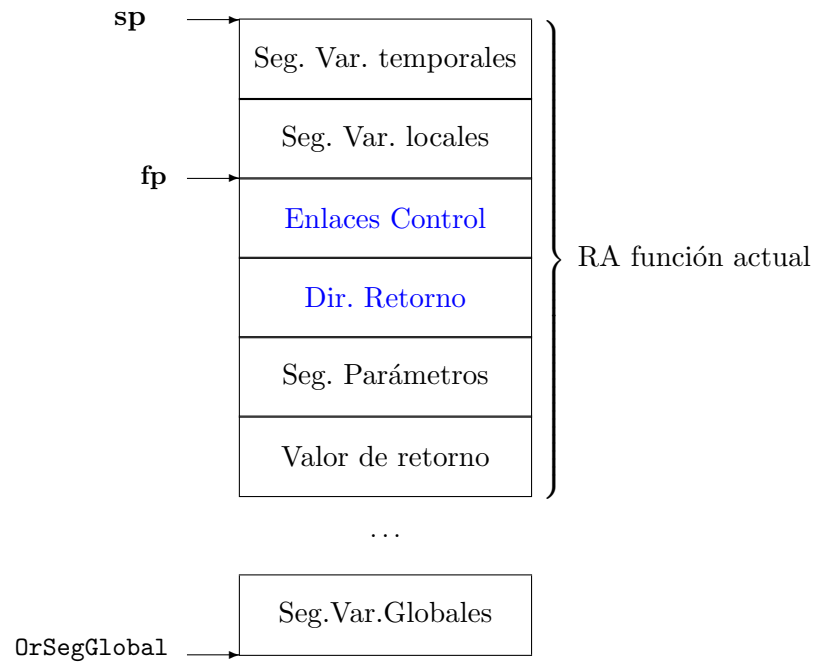


Figura 3: Estructura de un RA en Malpas.

9.2.1. Estructura de los RA

En la Figura 3 se muestra la estructura de un RA tal y como debe construirse para que se ejecute correctamente el código Malpas. Donde: `OrSegGlobal` es el origen del segmento de las variables globales, en realidad es el origen de la pila; `fp` es el puntero al RA (“frame”) actual; y `sp` es la cima de la pila.

9.2.2. Acceso a las variables en mvm

El acceso a las variables se realiza de forma homogénea tanto para objetos globales como locales y parámetros. 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 *nivel del bloque* y el *desplazamiento relativo* al segmento correspondiente.

Para calcular esta dirección física, `mvm` internamente sumará el desplazamiento relativo del objeto a la dirección base del área de datos globales `OrSegGlobal` (si el objeto es global) o a la dirección apuntada por el FP, *frame pointer*, (si es un objeto local o parámetro).

10. 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
- el atributo del argumento: nivel en el bloque y su desplazamiento relativo para posición; valor entero para entero y etiqueta; y vacío para nulo.

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

10.1. 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. De entre todas podemos destacar:

- **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 tres variables globales para gestionar el `niv` nivel de anidamiento GLOBAL o LOCAL y `dvar` desplazamiento relativo en el segmento de datos (ambas definidas en `libgts.h`) y `si` el desplazamiento 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 nivel;          /* Nivel (local o global) en caso de tipo posición */
    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()`.

10.2. Funciones de ayuda a la GCI

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

```

/***** Funciones para facilitar la tarea de generación de código */
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 n, int valor) ;
/* Crea el argumento de una instrucción tres direcciones de tipo posición
   con la información del nivel "n" y del desplazamiento 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". */

```

10.3. 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 */

```

11. Ejemplos ilustrativos

11.1. Generación de código intermedio

A continuación se muestra un ejemplo sencillo de generación de código intermedio para la expresión `Aditiva`. En el ejemplo hemos considerado para las expresiones dos atributos: tipo, `t`, y desplazamiento relativo, `d`; por supuesto, estos atributos deben definirse en la `%union` del principio del programa BISON. Además, 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'");

        $$ . d = creaVarTemp();
        /****** Expresión a partir de un operador aritmético */
        emite($2, crArgPos(niv, $1 . d), crArgPos(niv, $3 . d), crArgPos(niv, $$ . d));
    }
    ;
```

11.2. 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 factorial (int n)
{ int f;

    if (n <= 1) f=1;
    else f= n * factorial(n-1);
    return f;
}

int main ()
{ int x;

    read(x);
    if (x > 0)
        if (x < 13) print(factorial(x));
        else {}
    else {}
    return 0;
}
```

Y el código tres direcciones será:

0	INCTOP	,	, i: 0
1	GOTOS	,	, e: 30
2	PUSHFP	,	,
3	FPTOP	,	,
4	INCTOP	,	, i: 12
5	EASIG	p: (1, -3) ,	, p: (1, 1)
6	EASIG	i: 1 ,	, p: (1, 2)
7	EASIG	i: 1 ,	, p: (1, 3)
8	EMENEQ	p: (1, 1) , p: (1, 2) ,	e: 10
9	EASIG	i: 0 ,	, p: (1, 3)
10	EIGUAL	p: (1, 3) , i: 0 ,	e: 14
11	EASIG	i: 1 ,	, p: (1, 4)
12	EASIG	p: (1, 4) ,	, p: (1, 0)
13	GOTOS	,	, e: 25
14	EASIG	p: (1, -3) ,	, p: (1, 5)
15	EPUSH	,	, i: 0
16	EASIG	p: (1, -3) ,	, p: (1, 6)
17	EASIG	i: 1 ,	, p: (1, 7)
18	EDIF	p: (1, 6) , p: (1, 7) ,	p: (1, 8)
19	EPUSH	,	, p: (1, 8)
20	CALL	,	, e: 2
21	DECTOP	,	, i: 1
22	EPOP	,	, p: (1, 9)
23	EMULT	p: (1, 5) , p: (1, 9) ,	p: (1, 10)
24	EASIG	p: (1, 10) ,	, p: (1, 0)
25	EASIG	p: (1, 0) ,	, p: (1, 11)
26	EASIG	p: (1, 11) ,	, p: (1, -4)
27	TOPFP	,	,
28	FPPOP	,	,
29	RET	,	,

30	PUSHFP	,	,
31	FPTOP	,	,
32	INCTOP	,	, i: 10
33	EREAD	,	, p: (1, 0)
34	EASIG	p: (1, 0) ,	, p: (1, 1)
35	EASIG	i: 0 ,	, p: (1, 2)
36	EASIG	i: 1 ,	, p: (1, 3)
37	EMAY	p: (1, 1) , p: (1, 2) ,	e: 39
38	EASIG	i: 0 ,	, p: (1, 3)
39	EIGUAL	p: (1, 3) , i: 0 ,	e: 55
40	EASIG	p: (1, 0) ,	, p: (1, 4)
41	EASIG	i: 13 ,	, p: (1, 5)
42	EASIG	i: 1 ,	, p: (1, 6)
43	EMEN	p: (1, 4) , p: (1, 5) ,	e: 45
44	EASIG	i: 0 ,	, p: (1, 6)
45	EIGUAL	p: (1, 6) , i: 0 ,	e: 54
46	EPUSH	,	, i: 0
47	EASIG	p: (1, 0) ,	, p: (1, 7)
48	EPUSH	,	, p: (1, 7)
49	CALL	,	, e: 2
50	DECTOP	,	, i: 1
51	EPOP	,	, p: (1, 8)
52	EWRITE	,	, p: (1, 8)
53	GOTOS	,	, e: 54
54	GOTOS	,	, e: 55
55	EASIG	i: 0 ,	, p: (1, 9)
56	EASIG	p: (1, 9) ,	, p: (1, -3)
57	TOPFP	,	,
58	FPPOP	,	,
59	FIN	,	,