Procesadores de Lenguajes (2011/2012)

Construcción de un compilador para: `MenosC`.12

Parte I, II y III

28 de febrero de 2012

Índice

1. Introducción

Las practicas de la asignatura de *Procesadores del Lenguaje* para este segundo parcial están orientadas a la realización de un proyecto "real", donde el alumno puede 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 global mucho más real del funcionamiento de un compilador.
2. Constatar que el alumno habrá participado en la elaboración de un compilador completo que funciona "realmente".
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.

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 debe ser 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 tipo 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 el manual en linea de BISON que se encuentra en `PoliformaT` (menú recursos > material para prácticas).

1.2. Evaluación

Para aprobar la asignatura es condición necesaria obtener el APTO en la evaluación final de la práctica. La entrega de la práctica se realiza a través de una "*tarea*" en `PoliformaT`. La fecha límite de entrega para la revisión del proyecto es el **4 de Junio de 2012**. La evaluación final de la práctica se realizará en dos fases:

- **Evaluación del compilador**: Calificada como APTO ó NO-APTO. Se probará el compilador mediante los ejemplos de programas proporcionados, y se debe:
  - detectar todos los errores léxico, sintácticos (Parte I) y semánticos (Parte II) que aparezcan en los programas de prueba;
  - generar el código intermedio que funcione correctamente para todos los programas de prueba correctos (Parte III).
- **Evaluación individual**: En el mismo día que el examen de teoría, el 4 de junio de 2012, se realizará un examen práctico en el laboratorio. En dicho examen individual, el alumno deberá demostrar sus conocimientos modificando su proyecto para resolver un problema práctico planteado. Esta prueba supondrá el 30 % de la nota final de la asignatura.

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 boletines B2: *Introducción al* FLEX y B4: *Introducción al* BISON. En realidad, esta parte puede considerarse una extensión del ejercicio propuesto en el boletín B4.

En `PoliformaT` se dejará un conjunto de programas `MenosC` de prueba para la evaluación de esta parte del proyecto.

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. Solo los primeros 14 caracteres son significativos. Se deben distinguir entre mayúsculas y las minúsculas ("case sensitive").
- 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 1.
- Aunque puedan aparecer constantes enteras y reales en el programa fuente, todas las constantes numéricas deben convertirse (truncando en su caso) a enteras.
- El signo + (ó −) de las constantes numéricas se tratará como un símbolo léxico independiente.
- Los espacios en blanco, retornos de línea y tabuladores se ignoran.
- 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) para `MenosC` se usará la herramienta BISON. La especificación sintáctica completa para del lenguaje `MenosC` se define en la gramática de 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 aparecen en negrita y el símbolo **cte** representa una constante numérica entera (o real truncada).

programa	→ secuenciaDeclaraciones
secuenciaDeclaraciones	→ declaracion | secuenciaDeclaraciones declaracion
declaracion	→ declaracionVariable | declaracionFuncion
declaracionVariable	→ tipo **id** ; | tipo **id** [**cte**] ;
tipo	→ **int** | **struct** { listaCampos }
listaCampos	→ declaracionVariable | listaCampos declaracionVariable
declaracionFuncion	→ cabeceraFuncion bloque
cabeceraFuncion	→ tipo **id** (parametrosFormales)
parametrosFormales	→ ϵ | listaParametrosFormales
listaParametrosFormales	→ tipo **id** | tipo **id** , listaParametrosFormales
bloque	→ { declaracionVariableLocal listaInstrucciones }
declaracionVariableLocal	→ ϵ | declaracionVariableLocal declaracionVariable
listaInstrucciones	→ ϵ | listaInstrucciones instruccion
instruccion	→ { declaracionVariableLocal listaInstrucciones }
	→ instruccionExpresion | instruccionEntradaSalida
	→ instruccionSeleccion | instruccionIteracion
	→ instruccionSalto
instruccionExpresion	→ ; | expresion ;
instruccionEntradaSalida	→ **read** (**id**) ; | **print** (expresion) ;
instruccionSeleccion	→ **if** (expresion) instruccion **else** instruccion
instruccionIteracion	→ **for** (expresionOpcional ; expresion ; expresionOpcional) instruccion
expresionOpcional	→ ϵ | expresion
instruccionSalto	→ **return** expresion ;
expresion	→ expresionIgualdad | **id** operadorAsignacion expresion
	→ **id** [expresion] operadorAsignacion expresion
	→ **id** . **id** operadorAsignacion expresion
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	→ **id** [expresion] | **id** . **id** | **id** operadorIncremento
	→ **id** (parametrosActuales) | (expresion) | **id** | **cte**
parametrosActuales	→ ϵ | listaParametrosActuales
listaParametrosActuales	→ expresion | expresion , listaParametrosActuales
operadorAsignacion	→ `=` | `+=` | `-=`
operadorIgualdad	→ `==` | `!=`
operadorRelacional	→ `>` | `<` | `>=` | `<=`
operadorAditivo	→ `+` | `-`
operadorMultiplicativo	→ `*` | `/`
operadorIncremento	→ `++` | `--`
operadorUnario	→ `+` | `-`

Figura 1: Especificación sintáctica del lenguaje `MenosC`.12

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 Tabla de Bloques (TDB), y la gestión de memoria estática. Posteriormente, en una tercera parte, se implementará el generador de código intermedio.

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

- `libtds`. Librería con las operaciones para la manipulación de la TDS y la TDB. Esta librería la podemos encontrar:
  - *En los equipos de los laboratorios* `$HOME/asigDSIC/FI/pdl/`
  - *Desde casa, en* `<alien3.dsic.upv.es>` `/labos/asignaturas/FI/pdl/`

  En `pdl` aparecen dos directorios: `include` y `lib`, donde se sitúa respectivamente el fichero con la cabecera `libtds.h` y el objeto `libtds.a` de la librería. En la tercera parte del proyecto, en dichos directorios añadiremos la información de una nueva librería para la generación de código.

  Independientemente de que podáis trabajar en casa con vuestros computadores, es importante advertir que el código de vuestro compilador debe funcionar para la configuración/distribución que hay en los equipos de los laboratorios docentes (o en `<alien3.dsic.upv.es>`).
- *Programas de prueba.* Para comprobar el funcionamiento de esta segunda parte del compilador se dispondrán en `PoliformaT` de ficheros de prueba 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 se usarán constantes enteras. Si el analizador léxico encuentra una constante real se debe devolver su valor entero truncado.
- Todas las variables y funciones deben declararse antes de ser utilizadas.

```
#define MAX_LENGTH 14
#define  Devolver(x) if (verbosidad) ECHO; return x;
void creaNombre();
void creaCentera();
void truncCreal();
```

Figura 2: Ejemplo de preámbulo de usuario para el programa FLEX.

- La talla del tipo *entero* debe estar definida en la constante TALLA_ENTERO= 1.
- El tipo de los elementos de los vectores y los campos de los registros debe ser `int`.
- Tanto las funciones como sus parámetros solo pueden ser de tipo `int`.
- 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.
- Con respecto al alcance de las variables se siguen las restricciones semánticas propias de un Lenguaje con Estructura de Bloques.
- En todo programa debe haber una función (y solo una) denominada *main*, la cual determina el inicio y el fin en la ejecución del programa.
- El paso de parámetros se hace siempre por valor.
- Se permite la recursividad.
- Tanto la información de los parámetros como la memoria reservada para los mismos se hará en orden inverso a su declaración.
- La talla del área de enlaces de control (que se usará para la generación de código para los Registros de Activación) debe estar definida en la constante TALLA_SEGENLACES= 2.
- Las expresiones lógicas, que se pueden formar a partir de los operadores relacionales, deberán tener un valor 0, para el caso `falso`, y 1, para el caso `verdad`.
- En las instrucciones `if-else` ó `for`, las expresiones deben ser de tipo lógico.
- Los operadores de incremento ++ y decremento −− (infijos y postfijos), incrementan o decrementan en 1 el valor almacenado en una variable. Su semántica es la misma que la del lenguaje `C`.
- La asignación compuesta + = y − = permite realizar dos operaciones; por ejemplo, $a+=5$ es equivalente a $a = a + 5$.
- En cualquier otro caso, las restricciones semánticas por defecto serán las propias del lenguaje ANSI *C*.

4.1. Recomendaciones de implementación: atributos léxicos

Para la evaluación de los atributos léxicos asociados con los *identificadores*, *constantes enteras* y *constantes reales*, en el programa FLEX se deben definir unas funciones propias que se pueden situar en la sección de **funciones de usuario**. Para las restricciones léxicas definidas en la Sección 2, una posibilidad para esta sección podría ser la que se indica en la Figura 3. Advertid que también se deben definir los perfiles de dichas funciones en la subsección de **preámbulo-C**, como se indica en la Figura 2.

```
/*****************************************************************************/
void creaNombre()
/* Comprueba que el identificador no exceda la talla máxima (14); en ese      */
/* caso, lo trunca.                                                           */
{ char *t;
  if (yyleng > MAX_LENGTH) {
    if ( verbosidad ) fprintf(stdout,
      "Warning at line %d: identificador truncado a longitud %d\n",
      yylineno, MAX_LENGTH);
    yyleng=MAX_LENGTH;
  }
  t = (char *)malloc(yyleng+1); strncpy(t, yytext, yyleng);
  t[yyleng] = '\0'; yylval.ident = t;
}
/*****************************************************************************/
void creaCentera()
/* Transforma una subcadena a la constante entera que representa             */
{ yylval.cent = atoi(yytext); }
/*****************************************************************************/
void truncCreal()
/* Transforma una subcadena, que representa una constante real, a una        */
/* constante entera truncándola.                                             */
{ char *t;
  fprintf(stdout,
    "Warning at line %d: constante real truncada a entera\n", yylineno);
  yyleng = strcspn(yytext,".");
  t = (char *)malloc(yyleng+1); strncpy(t, yytext, yyleng);
  t[yyleng] = '\0'; yylval.cent = atoi(t);
}
/*****************************************************************************/
```

Figura 3: Ejemplo de funciones de usuario para el programa FLEX.

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.

El modo en el que se debe usar esta librería es:

```
gcc -o cmc alex.o asin.o -L./lib -I./include -lfl -ltds
```

Donde en `-I./include` le indicamos el directorio donde situaremos los distintos ficheros de cabecera, incluido el `libtds.h`; en `-L./lib` le indicamos el directorio donde situaremos los objetos de las librerías, incluido el `libtds.a`; y, finalmente, con `-ltds` le indicamos la librería a incluir. Es aconsejable que todo esto se adjunte en un fichero `makefile` para la correcta compilación de las distintas partes del proyecto (ver Sección 6.3).

5.1. Estructura de la Tabla de Símbolos (TDS)

La definición y manipulación de la TDS es la que corresponde a un Lenguaje con Estructura de Bloques (LEB). Esto repercute, entre otros, en dos importantes aspectos:

- **El alcance de las variables**.- Pueden aparecer varios objetos con el mismo nombre, siempre que estén definidos en bloques diferentes. En un cierto bloque se pueden utilizar los objetos declarados en dicho bloque y los declarados en bloques de niveles superiores accesibles.
- **La gestión de la TDS**.- La TDS se va a gestionar como una pila, de forma que mientras se esté analizando el bloque actual, sus objetos locales son accesibles en la TDS. Una vez finalizado el análisis de dicho bloque, sus objetos asociados dejarán de estar accesibles en la TDS (ver Sección 5.3).

En el fichero `libtds.h` aparece la definición de las constantes simbólicas, estructuras y cabeceras de funciones que serán de utilidad al implementar las acciones semánticas para manipular la TDS.

A modo ilustrativo, en la Figura 4 se muestran las estructuras básicas que contienen la información de la TDS para los objetos en general y para los vectores, registros y dominios de las funciones en particular. Posteriormente se verá que algunas de las funciones de consulta a la TDS devuelven estas estructuras.

```
typedef struct simb /******************** Elementos de la Tabla de Símbolos */
{
 int   categoria;                  /* Categoría del objeto                  */
 int   tipo;                       /* Tipo del objeto                       */
 int   desp;                       /* Desplazamiento relativo en memoria    */
 int   nivel;                      /* Nivel del bloque                      */
 int   ref;                        /* Campo de referencia de usos múltiples */
}SIMB;
typedef struct dim  /************************ Elementos de la Tabla de Array */
{
 int   tipo;                    /* Tipo de los elementos                    */
 int   lsup;                    /* Número de elementos del array            */
}DIM;
typedef struct inf  /***************** Información asociada a las funciones */
{
 char *nombre;                  /* Nombre de la funcion                     */
 int   tipo;                    /* Tipo devuelto por la función             */
 int   tparam;                  /* Numero (talla) de parámetros             */
}INF;
typedef struct reg  /************* Estructura para los campos de un registro */
{
  int   tipo;                         /* Tipo del campo                     */
  int   desp;                         /* Desplazamiento relativo del campo  */
}REG;
```

Figura 4: Estructuras básicas que contienen información de la TDS.

```
/****************************************** Tipos para la Tabla de Símbolos */
#define T_VACIO       0
#define T_ENTERO      1
#define T_LOGICO      2
#define T_ARRAY       3
#define T_RECORD      4
#define T_ERROR       5
/************************************* Categorías para la Tabla de Símbolos */
#define NULO          0
#define VARIABLE      1
#define FUNCION       2
#define PARAMETRO     3
```

Figura 5: Constantes generales usadas en `libtds`.

Finalmente, también es necesario considerar las constantes simbólicas definidas para representar los tipos y categorías de los objetos del lenguaje que se utilizan en la librería; un resumen de las más importantes se recogen en la Figura 5.

5.2. Funciones de manipulación de la TDS

En esta sección se presenta el listado de funciones que deben emplearse para completar y consultar la TDS, y sus tablas auxiliares.

```
int  insertaSimbolo(char *nom, int clase, int tipo, int desp, int n, int ref);
/* Inserta en la TDS toda la información asociada con un símbolo de: nombre
   "nom", clase "clase", tipo "tipo", desplazamiento relativo en el segmento
   correspondiente (variables, parámetros o instrucciones) "desp", nivel del
   bloque "n" y referencia a posibles subtablas "ref" (-1 si no referencia a
   otras subtablas). Si el identificador ya existe en el bloque actual,
   devuelve el valor "FALSE=0" ("TRUE=1" en caso contrario).                */
int  insertaInfoArray (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.                                        */
int  insertaInfoCampo (int refe, char *nom, int tipo, int desp);
/* Inserta en la Tabla de Registros, referenciada por "refe", la información
   de un determinado campo: nombre de campo "nom", tipo de campo "tipo" y
   desplazamiento del campo "desp". Si "ref = -1" entonces crea una nueva
   entrada en la Tabla de Registros para este campo y devuelve su referencia.
   Comprueba además que el nombre del campo no este repetido en el registro,
   devolviendo "-1" en caso de algún error.                                 */
int  insertaInfoDominio (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".       */
SIMB obtenerSimbolo (char *nom);
/* Obtiene toda la información asociada con un objeto de nombre "nom" y la
   devuelve en una estructura de tipo "SIMB". Si el objeto no está declarado,
   en el campo "categoria" devuelve el valor "NULO".                        */
```

```
DIM  obtenerInfoArray (int ref);
/* Devuelve toda la información asociada con un array referenciado por "ref"
   en la Tabla de Arrays.                                                    */
INF  obtenerInfoFuncion (int ref);
/* Devuelve la informacion del nombre de la función, el tipo del rango y el
   numero (talla) del segmento de parametros de una función cuyo dominio
   esta referenciado por "ref" en la TDS. Si "ref<0" entonces devuelve la
   informacion de la funcion actual.                                         */
int  comparaDominio (int refx, int refy);
/* Si los dominios referenciados por "refx" y "refy" no coinciden devuelve
   "FALSE=0" ("TRUE=1" si son iguales).                                      */
REG  obtenerInfoCampo (int ref, char *nom);
/* Obtiene toda la información asociada con un campo, de nombre "nom", de un
   registro referenciado por el índice "ref" en la Tabla de Registros. Si
   no se encuentra devuelve "T_ERROR" en el campo "tipo".                    */
void mostrarTDS (int n);
/* Muestra en pantalla toda la información de la TDS asociada con el bloque
   definido por "n".                                                         */
```

5.2.1. Ejemplo de uso de las tablas auxiliares

En varias de las funciones anteriores se puede ver como se crean referencias a tablas auxiliares. Veamos su utilización mediante un ejemplo. Supongamos que se desea introducir en la TDS una función con su dominio asociado. Para ello se deben hacer los siguientes pasos en las reglas sintácticas correspondientes:

1. Introducir en la TDS el tipo del primer parámetro. Como se trata del primer parámetro es necesario crear al mismo tiempo la referencia para el dominio. Para ello se debe poner un valor -1 en el argumento `refe` de la llamada a `insertaInfoDominio`. Esta llamada devolverá también la referencia creada.

   ```
   referencia = insertaInfoDominio(-1, tipo_del_parámetro);
   ```

2. Introducir mediante sucesivas llamadas a `insertaInfoDominio` los tipos del resto de los parámetros de la función, usando repetidamente la referencia devuelta para el primero:

   ```
   insertaInfoDominio(referencia, tipo_del_parámetro);
   ```

3. Finalmente, con la referencia al dominio de la función que nos proporcionó `insertaInfoDominio`, debemos almacenar esta información en el argumento `refe` de la función `insertaSimbolo` cuando insertemos la información de nuestra función en la TDS.

5.3. Funciones de manipulación de la Tabla de Bloques

La gestión de la TDS como una pila resulta mucho más eficiente si se apoya en una Tabla de Bloques. A continuación se muestran las dos funciones de que se dispone para gestionar esta tabla.

```
void cargaContexto (int n);
/* Crea el contexto necesario así como las inicializaciones de la TDS y
   la TDB para un nuevo bloque con nivel de anidamiento "n". Si "n=0"
   corresponde a los objetos globales; si "n=1" a los objetos locales a las
   funciones;  y si "n>0" a los objetos locales al nuevo bloque.            */
void descargaContexto (int n);
/* Libera en la TDB y la TDS el contexto asociado con el bloque "n".        */
```

5.3.1. Ejemplo del uso de la gestión de los bloques

A continuación se muestran un ejemplo de donde y cómo podría realizarse la llamada a estas funciones que manejan la Tabla de Bloques.

Programa	:	{ `nivel=0`; `cargaContexto(nivel)`; ... }
		secuenciaDeclaraciones
		{ ... `descargaContexto(nivel)`; }
cabeceraFuncion	:	tipo **id**
		{ `nivel++`; `cargaContexto(nivel)`; ... }
		(parametrosFormales)
		{ ... }
declaracionFuncion	:	cabeceraFuncion
		{ ... }
		bloque
		{ ... `descargaContexto(nivel)`; `nivel--`; }
instruccion	:	{ `nivel++`; `cargaContexto(nivel)`; ... }
		{ declaracionVariableLocal listaInstrucciones }
		{ ... `descargaContexto(nivel)`; `nivel--`; }

Donde `nivel` es una variable a definir para gestionar el nivel de anidamiento de los bloques.

6. Ejemplos y recomendaciones de implementación

En esta sección se muestran dos ejemplos sencillos de comprobación de tipos: uno en la declaración de variables y otro en el de las expresiones. Para terminar, se recopilan algunos consejos finales de implementación.

6.1. Comprobación de tipos en *declaraciones*

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

```
declaracion : declaracionVariable
  { if (!insertaSimbolo($1.nombre,VARIABLE,$1.tipo,dvar,nivel,$1.ref))
       yyerror ("Identificador repetido");
     dvar += $1.talla;
  }
declaracionVariable : tipo ID_ PUNTOCOMA_
  { $$.nombre=$2; $$.tipo=$1.tipo; $$.talla=$1.talla; $$.ref=$1.ref; }
tipo : INT_
  { $$.tipo=T_ENTERO; $$.talla=TALLA_ENTERO; $$.ref=-1; }
```

Donde `dvar` es una variable a definir para gestionar el desplazamiento relativo en el segmento de variables. Para `declaracionVariable` y `tipo` se ha definido un atributo registro con cuyos campos han sido: `nombre`, `tipo`, `talla` y `ref`.

6.2. Comprobación de tipos en *expresiones*

En el caso de una expresión, donde se espera que sea del tipo entero, su comprobación de tipos podría ser:

```
expresion : ID_ operadorAsignacion expresion
  { SIMB sim;
     sim = obtenerSimbolo($1);
     if (sim.categoría == NULO)
       yyerror("Identificador no declarado");
     if ((sim.tipo==$3.tipo)&&(sim.tipo==T_ENTERO)) $$=T_ENTERO;
     else {
       if (($3.tipo!=T_ERROR)&&(sim.tipo!=T_ERROR))
         yyerror("Error de tipos en la asignacion de la «expresion»");
       $$.tipo=T_ERROR;
     }
  }
```

Advertid que para evitar una secuencia de errores redundantes, solo se dará un nuevo mensaje de error si el error se produce en esta regla (y no si proviene de errores anteriores).

6.3. Recomendaciones de implementación

Para terminar sería interesante mencionar algunas recomendaciones para la correcta realización del proyecto:

- Las constantes, estructuras y variables globales que se utilicen en todo el compilador, sería conveniente definirlas en un fichero de cabecera `header.h` (y situarlo en el directorio `include`).

- Diseñad un programa principal que gestione la linea de comandos y permita invocar adecuadamente al compilador. Un posible ejemplo podría ser:

```
int main (int argc, char **argv)
/*  Programa principal: gestiona la linea de comandos e invoca al
    analizador  sintactico-semantico.                                  */
{ int i, n = 0;
  for (i=0; i<argc; ++i) {
    if (strcmp(argv[i], "-v")==0) { verbosidad = TRUE; n++; }
    else if (strcmp(argv[i], "-t")==0) { verTDS = TRUE; n++; }
  }
  --argc; n++;
  if (argc == n) {
    if ((yyin = fopen (argv[argc], "r")) == NULL)
      fprintf (stderr, "Fichero no valido %s\n", argv[argc]);
    else \{
      if (verbosidad == TRUE) fprintf(stdout,"%3d.- ", yylineno);
      yyparse ();
      if (numErrores > 0)
        fprintf(stdout,"\nNumero de errores:      %d\\n", numErrores);
    }
   }
   else fprintf (stderr, "Uso: cmc [-v] [-t] fichero\n");
}
```

  Como se puede apreciar, la invocación del compilador, por parte de un usuario, podrá tener, opcionalmente, 2 parámetros: `-v` si se desea verbosidad; y `-t` si se desea una visualización de la TDS. Estos parámetros activan unas variables globales (`verbosidad` y `ver_tds`) que, en este caso, hemos definido. También se utiliza una variable `numErrores` para contabilizar los posibles errores detectados.

- Análogamente es necesario definir una función para gestionar los posibles errores. Un ejemplo podría ser:

```
void yyerror(const char * msg)
/*  Tratamiento de errores.                                  */
{ numErrores++;
  fprintf(stdout, "Error at line %d: %s\n", yylineno, msg);
}
```

- Finalmente, se recomienda la creación de un fichero `makefile` para realizar correctamente la tarea de compilación, carga y edición de enlaces de las distintas partes del proyecto, con solo ejecutar el comando *make*. Un posible ejemplo podría ser:

```
cmc: alex.o  asin.o
        gcc -o cmc alex.o  asin.o -L./lib -I./include  -lfl -ltds
asin.o: asin.c
        gcc -c asin.c -I./include
alex.o: alex.c asin.c
        gcc -c alex.c -I./include
asin.c: asin.y
        bison -oasin.c -d asin.y
        mv asin.h ./include
alex.c: alex.l
        flex -oalex.c alex.l
```

Parte III
Generador de Código Intermedio

Teniendo en cuenta el trabajo desarrollado en las dos primeras fases del proyecto[1] 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 de una máquina virtual denominada `Malpas`.

Para facilitar el desarrollo de esta parte del compilador, se proporciona el siguiente material auxiliar:

- `libgci`. Librería con las operaciones para la generación de código intermedio. Como en el caso anterior, esta librería la podemos encontrar:
  - *En los equipos de los laboratorios* `$HOME/asigDSIC/FI/pdl/`
  - *Desde casa, en* `<alien3.dsic.upv.es>` `/labos/asignaturas/FI/pdl/`

  Como en el caso anterior, el fichero de cabeceras `libgci.h` se sitúa en el directorio `include` y la propia librería `libtds.a` en el directorio `src`
- `Malpas`. Máquina virtual para la ejecución del código intermedio generado por vuestro compilador. Como en el caso de las librerías, esta máquina está disponible en un directorio de nombre `bin`.
- *Programas de prueba.* Para comprobar el funcionamiento de esta segunda parte del compilador se dispondrán en `PoliformaT` de ficheros de prueba 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 `Malpas`.

La entrega final del proyecto se hará completando la *"tarea"* correspondiente en `PoliformaT`. En dicha tarea se deberá indicar el nombre de todos los miembros del grupo y se tendrá que adjuntar los ficheros que constituyen vuestro compilador (p.ej. `alex.l`, `asin.y` y `header.h`).

[1] **Fe de erratas**:

- `Sec.5.2`: se ha eliminado la función `esMain`
- `Sec.5.2`: se ha añadido un nuevo campo (en rojo) `nombre` al tipo `INF` en la Figura 4 que nos ayudará a verificar si la función actual es "main". Con este propósito, también se ha modificado (en rojo) la función `obtenerInfoFuncion`. Ambas modificaciones se han incorporado a la librería `libtds` y por tanto deberéis actualizarla.
- `Sec.6.2`: se han corregido (en rojo) una errata en el ejemplo.

7. 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 (intermedio tres direcciones) para una máquina (virtual) denominada `Malpas`.

7.1. Inventario de instrucciones tres direcciones (`Malpas`)

En esta sección se presenta el juego de instrucciones de la máquina virtual `Malpas`, agrupadas por categorías. Para cada instrucción se definen cuatro partes: *código de operación* (OP); dos *argumentos* (arg1 y arg2) y un *resultado* (res). Además se proporciona una pequeña leyenda con su *significado*.

Los argumentos y el resultado pueden ser: *enteros I*; *posición P*;*etiquetas E* y *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
ETOB	I/P		P	Convierte entero a lógico
BTOE	I/P		P	Convierte lógico a entero

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 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 *res*

Operaciones de entrada/salida

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

Operaciones de manejo de pila de RA

OP	arg1	arg2	res	Significado
EPUSH			I/P	Apila un elemento en la cima de la pila
EPOP			P	Desapila la cima y la deposita en *res*
PUSHFP				Apila el frame pointer
FPPOP				Desapila la cima y la deposita en el frame pointer
FPTOP				El frame pointer apunta a la posición apuntada por el tope de la pila
TOPFP				El tope de la pila apunta a la posición apuntada por el frame pointer
INCTOP			I	Incrementa el tope de la pila en *res* posiciones
DECTOP			I	Decrementa el tope de la pila en *res* posiciones

7.2. Arquitectura de `Malpas`

La memoria de `Malpas` 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).

Dado que `MenosC` no permite objetos dinámicos, la gestión de memoria de `Malpas` se reducirá pues a la manipulación de esta pila de RA.

7.2.1. Estructura de los RA

En la Figura 6 se muestra la estructura de un RA tal y como debe construirse para que se ejecute correctamente el código en `Malpas`.

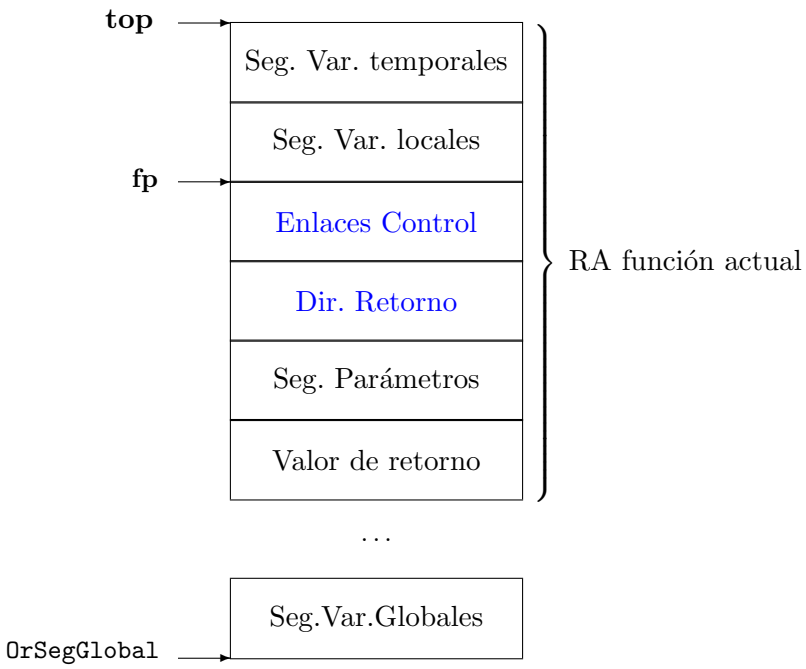

Figura 6: Estructura de un RA en `Malpas`.

7.2.2. Acceso a las variables en el RA

El acceso a las variables se realiza de forma homogénea tanto para objetos globales como locales. La posición de memoria asignada a un objeto global se caracteriza por tener un *nivel de bloque* $=$ 0, mientras que los objetos locales a las funciones tendrán un *nivel de bloque* $\geq$ 1.

El acceso a la dirección absoluta de memoria donde se encuentra un objeto es responsabilidad de `Malpas`, debiendo proporcionarle (en el argumento correspondiente de la instrucción de código 3 direcciones): el *nivel del bloque* y el *desplazamiento relativo* al segmento correspondiente. Para calcular esta posición absoluta `Malpas` 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 "frame pointer" (si es un objeto local).

8. Generación de código intermedio

Al finalizar la compilación de un programa fuente se debe de volcar el código intermedio a un fichero de texto. Este volcado se realizará siempre y cuando no se hayan detectado errores en la compilación.

Las instrucciones de código intermedio 3 direcciones tienen el siguiente formato:

```
<num_linea>  <COD_OPERACION>  <arg1>  <arg2>  <res>
```

Donde para cada uno de los argumentos ($arg1$, $arg2$ y res) se debe indicar: el tipo del argumento (**p**: posición, **i**: entero, **e**: etiqueta) y su valor. En el caso de los argumentos de tipo posición se debe indicar además su *nivel de bloque* y su *desplazamiento relativo*.

8.1. Estructuras de datos y variables globales

En el fichero de cabecera `libgci` se encuentran la definición de las estructuras, las constantes simbólicas, las variables globales y las cabeceras de las funciones necesarias para la generación de código tres direcciones. De entre todas, podemos destacar las siguientes declaraciones útiles:

```
/********* 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
typedef struct tipo_pos /*********** Estructura para una posición de memoria */
{
  int pos;                            /*     Posicion relativa de memoria    */
  int niv;                            /*     Contexto (nivel) de la variable */
} TIPO_POS;
typedef struct tipo_arg /****** Estructura para los argumentos del codigo 3D */
{
  int tipo;                  /* Tipo del argumento                           */
  union {
    int      i;              /* Variable para argumento entero               */
    TIPO_POS p;              /* Variable para argumento posicion de memoria  */
    int e;                   /* Variable para argumento direccion de memoria */
  } val;
} TIPO_ARG;
/************************** Variables globales de uso en todo el compilador */
extern int si;                /* Desplazamiento en el Segmento de Codigo     */
extern int dvar;              /* Desplazamiento en el Segmento de Variables  */
```

Como se puede apreciar, esta librería define y maneja dos variables globales para gestionar el desplazamiento relativo: **dvar**[2] en el segmento de datos, y **si** en el segmento de código.

TIPO_ARG es un tipo que se corresponde con una estructura que contiene en su elemento `val` una unión. De esta forma, se podrá almacenar la información de un argumento de cualquier tipo. Por ejemplo, para almacenar un argumento de tipo entero con valor 5, basta con realizar las asignaciones `arg1.tipo`=ARG_ENTERO; `arg1.val.i`= 5. No obstante, no recomendamos realizar este tipo de asignaciones, ya que se dispone de *funciones*

[2]Es posible que en la segunda parte de vuestro proyecto hayáis definido una variable similar. Advertid que debéis cambiar esto, ya que solo puede haber una y esta debe ser la `dvar` de `libgci`.

(presentadas en la siguiente sección) que permiten hacerlo de una manera mucho más sencilla: `crArgNulo()`, `crArgEntero()`, `crArgEtiqueta()` y `crArgPosicion()`.

8.2. Funciones de ayuda a la GCI

Se presenta a continuación el listado de funciones que deben emplearse para generar código intermedio.

```
void emite (int cop, TIPO_ARG arg1, TIPO_ARG arg2, TIPO_ARG res);
/* Crea una instrucción tres direcciones con el código de operación "cod" 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 entera (de talla "1"), en el segmento de
    variables (indicado por "dvar") del RA actual y devuelve su
    desplazamiento relativo. A continuación, incrementa "dvar".            */
void vuelcaCodigo (char *nom);
/* Vuelca el código generado, en modo texto, a 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 crArgNulo ();
/* Crea un argumento de una instrucción tres direcciones de tipo nulo.     */
TIPO_ARG crArgEntero (int valor);
/* Crea un argumento de una instrucción tres direcciones de tipo entero
   con la información de la constante entera dada en "valor".              */
TIPO_ARG crArgEtiqueta (int valor);
/*  Crea el argumento de una instrucción tres direcciones de tipo etiqueta
    con la información de la dirección dada en "valor".                     */
TIPO_ARG crArgPosicion (int n, int valor);
/*  Crea el argumento de una instrucción tres direcciones de tipo posición
    con la información del nivel en "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".                                            */
```

8.3. Ejemplo de producciones generadoras de código

Para que el alumno pueda hacerse una idea de la forma en la que se puede generar código empleando las funciones anteriores, se presentan a continuación algunas producciones con sus acciones semánticas de generación de código intermedio. El objetivo de estas producciones es generar el código intermedio para evaluar una expresión unaria (se ha eliminado, por tanto, la parte de comprobaciones semánticas.

```
operadorIncremento
       : MASMAS_       {   $$ = ESUM;  }
       | MENOSMENOS_   {   $$ = EDIF;  }
       ;

expresionUnaria : operadorIncremento ID_
        { SIMB sim;  TIPO_ARG res;

          sim = obtener_simbolo($2);
          /*       comprobaciones semánticas       */
          $$.tipo = T_ENTERO;
          res = crArgPosicion(sim.nivel, sim.desp);
          $$.pos = crArgPosicion(nivel, creaVarTemp());
          /************************************ INCREMENTA o DECREMENTA 1 */
          emite($1, res, crArgEntero(1), res);
          /************************************************** Asignación */
          emite(EASIG, res, crArgNulo(), $$.pos);
        }
```

Donde **nivel** es una variable que se debe definir para gestionar el nivel de anidamiento de los diferentes bloques.

8.4. Recomendaciones finales de implementación

Para terminar sería interesante mencionar un par de recomendaciones para la correcta generación de código intermedio.

- Se debe modificar el programa principal para que pueda volcar el código generado en el proceso de compilación. Un posible ejemplo podría ser:

```
int main (int argc, char **argv)
/*  Programa principal: gestiona la linea de comandos e invoca al
/*  analizador  sintactico-semantico.                              */
{ char *nom_fich; int i, n = 0;
  for (i=0; i<argc; ++i) {
    if (strcmp(argv[i], "-v")==0) { verbosidad = TRUE; n++; }
    else if (strcmp(argv[i], "-t")==0) { verTDS = TRUE; n++; }
  }
  --argc; n++;
  if (argc == n) {
    if ((yyin = fopen (argv[argc], "r")) == NULL)
      fprintf (stderr, "Fichero no valido %s\n", argv[argc]);
    else {
      if (verbosidad == TRUE) fprintf(stdout,"%3d.- ", yylineno);
      nom_fich = argv[argc];
      yyparse ();
      if (numErrores == 0) vuelcaCodigo(nom_fich);
```

```
      else fprintf(stdout,"\nNumero de errores:  %d\n", numErrores);
    }
  }
  else fprintf (stderr, "Uso: cmc [-v] [-t] fichero\n");
}
```

Donde la única diferencia respecto al ejemplo de la segunda parte radica en la variable `nom_fich` y su uso con la función `vuelcaCodigo(nom_fich)`, solo cuando no hay errores.

- Finalmente, se recomienda la modificación del fichero `makefile` para incluir la nueva librería. Un posible ejemplo podría ser:

```
cmc: alex.o  asin.o
        gcc -o cmc alex.o  asin.o -L./lib -I./include  -lfl -ltds -lgci
```

9. Ejemplo de 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 múmero < 20
int factorial(int n)
{
  if (n <= 1) return 1;
  else return n * factorial(n-1);
}
int main()
{ int x;
  read(x);
  if (x > 0)
    if (x < 20) print(factorial(x));
    else ;
  else ;
}
```

Y el código tres direcciones será:

0	INCTOP		,		,	i: 0
1	GOTOS		,		,	e: 24
2	PUSHFP		,		,	
3	FPTOP		,		,	
4	INCTOP		,		,	i: 4
5	EASIG	i: 1	,		,	p: (1,0)
6	EMENEQ	p: (1,-3)	,	i: 1	,	e: 8
7	EASIG	i: 0	,		,	p: (1,0)
8	EIGUAL	p: (1,0)	,	i: 0	,	e: 12
9	EASIG	i: 1	,		,	p: (1,-4)
10	GOTOS		,		,	e: 21
11	GOTOS		,		,	e: 21
12	EPUSH		,		,	i: 0
13	EDIF	p: (1,-3)	,	i: 1	,	p: (1,1)
14	EPUSH		,		,	p: (1,1)
15	CALL		,		,	e: 2
16	DECTOP		,		,	i: 1
17	EPOP		,		,	p: (1,2)
18	EMULT	p: (1,-3)	,	p: (1,2)	,	p: (1,3)
19	EASIG	p: (1,3)	,		,	p: (1,-4)
20	GOTOS		,		,	e: 21
21	TOPFP		,		,	
22	FPPOP		,		,	
23	RET		,		,	
24	PUSHFP		,		,	
25	FPTOP		,		,	
26	INCTOP		,		,	i: 4
27	EREAD		,		,	p: (1,0)
28	EASIG	i: 1	,		,	p: (1,1)
29	EMAY	p: (1,0)	,	i: 0	,	e: 31
30	EASIG	i: 0	,		,	p: (1,1)
31	EIGUAL	p: (1,1)	,	i: 0	,	e: 44
32	EASIG	i: 1	,		,	p: (1,2)
33	EMEN	p: (1,0)	,	i: 20	,	e: 35
34	EASIG	i: 0	,		,	p: (1,2)
35	EIGUAL	p: (1,2)	,	i: 0	,	e: 43
36	EPUSH		,		,	i: 0
37	EPUSH		,		,	p: (1,0)
38	CALL		,		,	e: 2
39	DECTOP		,		,	i: 1
40	EPOP		,		,	p: (1,3)
41	EWRITE		,		,	p: (1,3)
42	GOTOS		,		,	e: 43
43	GOTOS		,		,	e: 44
44	TOPFP		,		,	
45	FPPOP		,		,	
46	FIN		,		,	