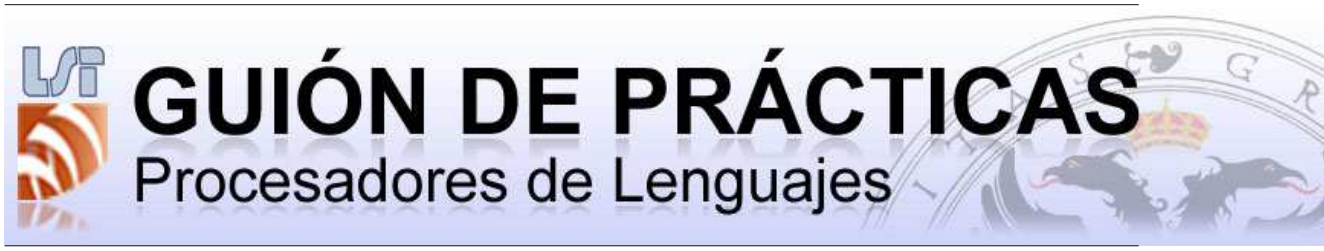

Curso 2020/2021

Autores del guión de Prácticas

Dr.Jorge Revelles Moreno

Dr.Carlos Ureña Almagro

Dr.Salvador Villena Morales

Control de Revisiones

06-Mar-2006	Versión inicial.
10-Mar-2006	Cambios en los siguientes apartados: 1. Apéndice B (YACC): Modificados los ejemplos LEX y YACC. 2. Apéndice C (Consideraciones sobre A. Sintáctico): Modificada la sección de informe de errores usando bison en lugar de yacc/byacc. También se incluye la traza de ejecución con sus informes de errores usando el ejemplo del apéndice B.
14-Mar-2006	Apéndice B (YACC): Modificado parte de los ejemplos LEX y YACC. Una parte corregía ciertos errores sintácticos en la especificación del ejemplo `prueba.y` y otros para cambios relativos a `prueba.l`.
13-Feb-2007	Cambios en los siguientes apartados: 1. Detallado aspectos de especificación del lenguaje en tipos estructurados (práctica 1). 2. Detallado error en declaración de variables (práctica 3). 3. Detallado conversión de tipos (práctica 4). 4. Corrección de errores en práctica 5. 5. Añadido opción mensajes de error con mes y mes2 (apéndice c). 6. Corrección de errores en apéndice d. 7. Acciones semánticas en YACC en lugar de en LEX (apéndice e).
19-Feb-2007	Añadido Apéndice con sintaxis y semántica de tipos estructurados
22-Feb-2007	Añadido en apéndice la precedencia y asociatividad de los operadores comunes
24-Feb-2007	Cambios de aspecto en práctica de generación de código
08-Mar-2007	Corrección de erratas en Práctica 1 (especificación del tipo de dato estructurado) y cambio en el símbolo de constantes de tipo conjunto
11-Feb-2008	Cambiadas las puntuaciones de las prácticas dado que la primera parte de la práctica de Generación de Código Intermedio es obligatoria desde el curso 2007/2008.
18-Feb-2008	Añadido Apéndice con el proceso de construcción de traductores usando Lex y Yacc para Java.
21-Feb-2008	Añadido ejemplo de código para práctica de Generación de Código Intermedio.
05-Mar-2008	Corrección de error en la especificación BNF de la práctica 1.
10-Feb-2009	Añadir mecanismo de paso de parámetros, eliminación de puntuación mínima en la práctica y corrección de errores en ejemplos de agregados.
17-Feb-2010	Correcciones mínimas y cambio de formato en algunos detalles

UNIVERSIDAD DE GRANADA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
DEPARTAMENTO DE LENGUAJES Y SISTEMAS INFORMÁTICOS

Guión de Prácticas de PROCESADORES DE LENGUAJES

Objetivo de las Prácticas

El objetivo de estas prácticas es familiarizarse con la implantación de traductores y en concreto con la implantación de compiladores haciendo uso de las herramientas LEX y YACC.

- **Práctica 1**. Diseño de un lenguaje de programación orientado al problema propuesto por el profesor.
- **Práctica 2**. Diseño e implantación de la fase de análisis de léxico usando el LEX.
- **Práctica 3**. Obtención de la gramática abstracta y la especificación YACC correspondiente, así como la implantación del analizador sintáctico.
- **Práctica 4**. Implantación de la tabla de símbolos y acciones semánticas, lo que constituye el analizador semántico
- **Práctica 5**. Generación de código intermedio basado en cuartetos con sintaxis de C (dispone de una parte obligatoria y dos partes optativas).

En cada práctica aparecen los objetivos, tareas a realizar y la documentación necesaria. **En la prueba de las prácticas 2, 3, 4 y 5 deberá estar actualizada e impresa la especificación del lenguaje en BNF y la descripción semántica del lenguaje (sobre todo, las operaciones con el tipo de dato específico y con el suficiente detalle)**.

Prácticas

Práctica 1

Diseño del Lenguaje

1.1 Objetivos

Obtener la definición del lenguaje de programación para el cual se va a diseñar el traductor.

1.2 Características del Lenguaje a Diseñar

El lenguaje a implementar será asignado por el profesor de prácticas y tendrá las siguientes características mínimas:

- Ser un subconjunto de un lenguaje de programación estructurado.
- Los identificadores debe ser declarados antes de ser usados.
- Los tipos de datos mínimos son: *entero*, *real*, *carácter* y *booleano*. Se definirán las operaciones típicas para cada uno de ellos, según se puede ver en la siguiente tabla.

Tipo de dato	Operaciones
entero, real	suma, resta, producto, división, operaciones de relación
booleano	and, or, not, xor

- Poseerá la sentencia de asignación para todos los tipos de expresiones.
- Permitirá expresiones aritméticas lógicas.
- Tendrá una sentencia de *entrada* y otra de *salida* (se utilizará como dispositivo de entrada el teclado y de salida la pantalla). Además, la sentencia de entrada deberá permitir leer sobre una lista de identificadores y la sentencia de salida deberá permitir escribir una **lista de expresiones y/o constantes de tipo cadena**. A diferencia de los lenguajes conocidos y usados como referencia, estas sentencias no representan llamada a subprograma.
- Dispone de las estructuras de control siguientes:
  - IF-THEN-ELSE.
  - WHILE.

- Con independencia del tipo de lenguaje asignado:
  - La estructura sintáctica del programa es:

```
<Programa> ::= <Cabecera_programa> <bloque>
```

  - En cualquier parte se podrán definir bloques como en C, es decir, tendremos una estructura sintáctica como la que se muestra a continuación:

```
<bloque> ::= <Inicio_de_bloque>
             <Declar_de_variables_locales>
             <Declar_de_subprogs>
             <Sentencias>
             <Fin_de_bloque>
```

  - Una sentencia puede ser, un bloque, por lo que se permite el anidamiento de bloques y subprogramas.
  - La comprobación de tipos será como la del Pascal, es decir, fuertemente tipado.
  - Para los argumentos de un subprograma, el mecanismo de paso de parámetros es por valor.
  - No se permiten declaraciones fuera de los bloques. Las declaraciones deben ir entre una marca de inicio y otra de final de las declaraciones[1].
  - La estructura sintáctica de un subprograma será el siguiente:

```
<Declar_subprog> ::= <Cabecera_subprograma> <bloque>
```

  - El lenguaje debe admitir tanto las letras mayúsculas como las minúsculas, exceptuando aquellos que tengan lenguaje C que ofrece sensibilidad en este sentido, mientras que para el caso de Pascal no sucede.

El lenguaje diseñado consecuente con los apartados anteriores deberá tener una estructura, empleando BNF, como sigue:

```
                  <Programa> ::= <Cabecera_programa> <bloque>
                    <bloque> ::= <Inicio_de_bloque>
                                 <Declar_de_variables_locales>
                                 <Declar_de_subprogs>
                                 <Sentencias>
                                 <Fin_de_bloque>
        <Declar_de_subprogs> ::= <Declar_de_subprogs> <Declar_subprog>
                                 |
            <Declar_subprog> ::= <Cabecera_subprograma> <bloque>
<Declar_de_variables_locales> ::= <Marca_ini_declar_variables>
                                 <Variables_locales>
                                 <Marca_fin_declar_variables>
                                 |
         <Cabecera_programa> ::= (Dependerá del lenguaje de referencia)
          <Inicio_de_bloque> ::= En C: {
                                 En Pascal: begin
             <Fin_de_bloque> ::= En C: }
                                 En Pascal: end
         <Variables_locales> ::= <Variables_locales> <Cuerpo_declar_variables>
                                 |  <Cuerpo_declar_variables>
    <Cuerpo_declar_variables> ::= (Dependerá del lenguaje de referencia)
           <Cabecera_subprog> ::= (Dependerá del lenguaje de referencia)
                 <Sentencias> ::= <Sentencias> <Sentencia>
                                 |  <Sentencia>
                  <Sentencia> ::= <bloque>
                                 |  <sentencia_asignacion>
                                 |  <sentencia_if>
                                 |  <sentencia_while>
```

[1] Esta restricción se impone para evitar, en algunos lenguajes, un conflicto en la tabla de análisis a la hora de generar el analizador sintáctico.

```
                            |  <sentencia_entrada>
                            |  <sentencia_salida>
                            |  <sentencia_return> (si el lenguaje soporta funciones)
                            |  <llamada_proced> (si el lenguaje soporta proced.)
                            |  (Resto de sentencias del lenguaje asignado)
      <sentencia_asignacion> ::= (Dependerá del lenguaje de referencia)
              <sentencia_if> ::= (Dependerá del lenguaje de referencia)
           <sentencia_while> ::= (Dependerá del lenguaje de referencia)
         <sentencia_entrada> ::= <nomb_entrada> <lista_variables>
          <sentencia_salida> ::= <nomb_salida> <lista_expresiones_o_cadena>
                 <expresion> ::= ( <expresion> )
                            |  <op_unario> <expresion>
                            |  <expresion> <op_binario> <expresion>
                            |  <identificador>
                            |  <constante>
                            |  <funcion> (si el lenguaje soporta funciones)
                            |  (Resto de expresiones del lenguaje de referencia)
```

El diseño del lenguaje asignado consiste en desarrollar los símbolos no terminales dependientes del lenguaje y consecuentes con los requisitos generales descritos anteriormente y con los requisitos particulares que se describirán a continuación.

El lenguaje debe incluir **cinco elementos nuevos**, además de los enumerado anteriormente. La generación de estos cinco elementos se generará de forma aleatoria y formará parte del grupo de prácticas.

1	2	3	4	5

A continuación se describe cada valor:

1. **Sintaxis inspirada en un lenguaje de programación:**

   (A) Pascal.

   (B) Lenguaje C.

   Esto significa que tomaremos las reglas sintácticas usadas por el lenguaje Pascal o C como referencia para las instrucciones del lenguaje nuevo, respetando en todo momento los requerimientos impuestos al lenguaje. Sin embargo, el lenguaje diseñado requiere que las declaraciones estén enmarcadas, mientras que en los lenguajes de referencia no se permite enmarcar las declaraciones. El profesor de prácticas informará al grupo de prácticas de las consideraciones sobre sintaxis en cada uno de los lenguajes a la hora de especificarlo en BNF.

2. **Palabras reservadas en:**

   (A) Castellano.

   (B) Inglés.

3. **Estructura de datos considerada como tipo elemental:** A continuación se definen los cuatro tipos de estructuras de datos que debe considerarse como si fuese un tipo elemental (entero, real, etc.) con sus operaciones. Sólo una de ellas es asignada por grupo y no se permite la declaración de variables de tipo estructurado de manera recursiva o mezclada, es decir, una lista de listas, lista de arrays, pila de listas, etc.

   (A) **Listas** con las operaciones para manejo de listas. Deben contemplarse las constantes de tipo lista (para mayor detalle del tipo estructurado, ver página 27 del apéndice A).

(B) **Array 1D y 2D** con las operaciones de acceso a un elemento, producto, suma y resta elemento a elemento, producto externo (producto de un array por un escalar) y producto de matrices, $C = A \times B$ sabiendo que el número de columnas de A debe coincidir con el número de filas de B. La matriz resultante C tendrá el número de filas de A y el número de columnas de B. Deben contemplarse las constantes de tipo array (para mayor detalle del tipo estructurado, ver página 28 del apéndice A).

(C) **Conjuntos** con las operaciones para manejo de conjuntos. Debe definirse la constante de tipo conjunto y un término para indicar el conjunto vacío (para mayor detalle del tipo estructurado, ver página 30 del apéndice A).

(D) **Pilas** con las operaciones de manejo de datos de tipo pila. Deben contemplarse las constantes de tipo pila (para mayor detalle del tipo estructurado, ver página 32 del apéndice A).

Los elementos de las estructuras anteriores podrán ser sólo del tipo básico definido: entero, real, carácter o booleano a excepción de los conjuntos que, tal y como se indica, sólo pueden ser de tipo entero o de tipo carácter. La sintaxis del lenguaje diseñado deberá permitir construir expresiones complejas con las operaciones definidas en cada estructura y, siempre que sea posible, dichas operaciones deben estar integradas en las expresiones aritmético-lógicas, permitiendo operaciones entre operandos de tipo simple y de tipo estructurado del mismo tipo simple.

4. **Subprogramas:**

   (A) Funciones.

   (B) Procedimientos.

5. **Estructuras de control adicional:**

   (A) case/switch.

   (B) for (usando para ello la misma sintaxis de Pascal).

   (C) repeat-until.

   (D) do-until.

1.3 Documentación a Presentar

La documentación a presentar debe contener la siguiente información:

1. Introducción, que contendrá una breve descripción del lenguaje asignado.

2. Descripción formal de la sintaxis del lenguaje usando **BNF**.

3. Definición de la semántica en lenguaje natural. Consiste en una descripción sencilla pero exacta de como se ejecuta cada instrucción del lenguaje asignado en los términos necesarios para que un compañero de otro grupo pudiera entenderlo y pudiera expresar algoritmos en vuestro lenguaje. Por ejemplo, se debería describir con detalle el significado de las operaciones con las estructuras compuestas, pero no sería necesario explicar como funciona la instrucción `if`.

4. Identificación de los **tokens** con el máximo nivel de abstracción. Esta labor será primordial para la realización de las siguientes prácticas.

Esta práctica será validada cuando cumpla todos los requisitos del lenguaje a ser diseñado y hayan sido identificados los tokens con el máximo nivel de abstracción.

Como consideración final, es posible utilizar la herramienta de apoyo Kaeru, que se expone en el **apéndice G** (ver página 61).

Práctica 2

Especificación e Implementación del Analizador de Léxico

2.1 Objetivos

Obtener el analizador de léxico para el lenguaje definido en la práctica anterior. La especificación léxica se obtendrá en base a la tabla de tokens que deberá construirse.

2.2 Proceso de Especificación del Analizador de Léxico

1. Confeccionar la tabla de tokens con el máximo nivel de abstracción para lenguaje diseñado en la práctica anterior con la información sigiente:

Nombre del token	Expresión regular	Código del token	Atributos

   Para obtener los tokens[1] con el máximo nivel de abstracción tenemos que asegurarnos que no se puede definir otro token que incluya a tokens ya definidos previamente. Cuanto más alto sea el nivel de abstracción aplicado para definir los tokens, menor será la complejidad en la fase de análisis léxico y sintáctico.

2. Confeccionar la especificación LEX [Lesk75, Levi92] que reconozca los tokens obtenidos en el apartado anterior. Por defecto, cuando se detecta un error en el análisis de léxico, la recuperación del mismo se produce volviendo al estado inicial despreciando la cadena de símbolos donde ha surgido el error.

2.3 Documentación a Presentar

La realización de la práctica 2 debe ir acompañada de la siguiente documentación:

1. Documentación de la primera práctica.

[1] Conviene otorgar un código al token a partir del valor 256 en adelante para seguir el estándar de LEX, donde los anteriores valores se asignan automáticamente siguiendo la codificación ASCII.

2. Tabla de tokens (incluyendo atributos, en su caso).
3. Especificación LEX.

2.4 Prueba de la Práctica

Para superar esta práctica el profesor comprobará que satisface las siguientes pruebas en su totalidad:

1. Tokens con el máximo nivel de abstracción atendiendo a la misión sintáctica.
2. Dado un programa expresado en el lenguaje diseñado y confeccionado por los alumnos integrantes del grupo de prácticas, el analizador de léxico deberá obtener la secuencia de código de token, atributo (en su caso) y lexema
3. Sobre el programa ejemplo de la prueba anterior, el profesor introducirá errores de léxico, debiendo ser detectados todos los errores introducidos y por consiguiente el léxico deberá recuperarse ante los errores.

Consideraciones de Portabilidad Windows/Linux

Para aquellos que trabajen indistintamente en Windows y Linux han de saber que la descripción LEX es totalmente compatible con Linux y Windows. Una breve información sobre FLEX se encuentra en el **apéndice B** (ver página 35).

Lo único que ha de tener en cuenta es que los archivos deben ser previamente convertidos al formato adecuado. Cuando se trabaja en Windows y se desea pasar a Linux, antes debe convertirse de formato DOS a UNIX (`dos2unix` o `iconv`, que es más completa que la anterior). Para el paso inverso, es decir, convertir de UNIX a DOS (`unix2dos`).

También es posible que existan problemas a la hora de codificar los caracteres. Se recomienda usar una única codificación en todos los archivos para evitar problemas de compatibilidad en las especificaciones Lex/Yacc.

Práctica 3

Analizador Sintáctico

3.1 Objetivos

Obtener un analizador sintáctico para el lenguaje definido en la primera práctica.

3.2 Tareas a Realizar

1. Definir la gramática abstracta del lenguaje, a partir de la especificación BNF obtenida en la primera práctica y teniendo en cuenta la tabla de tokens con el máximo nivel de abstracción.
2. Convertir la definición anterior en un fichero de entrada YACC [Levi92]. Procesar este fichero con el generador YACC/BYACC/BISON hasta que hayamos eliminado todos los conflictos reduce/reduce y shift/reduce a excepción de un conflicto del último tipo correspondiente a la construcción else opcional (ver información adicional).
3. Unir este analizador sintáctico obtenido con YACC/BYACC/BISON con el analizador léxico resultado de FLEX (en la segunda práctica), obteniéndose un programa ejecutable que implementa el analizador sintáctico. Comprobar este analizador con ejemplos de programas fuente escritos en nuestro lenguaje (ver información adicional).

3.3 Documentación a Presentar

La documentación a presentar para la evaluación de esta práctica consistirá en:

1. Documentación de la primera práctica.
2. Tabla de tokens (incluyendo atributos, en su caso).
3. Especificación LEX con los cambios adecuados para integrarla en el analizador sintáctico, sobre todo, en cuanto a consideraciones de los operadores em cuanto a asociatividad y precedencia (ver sección D.1 del apéndice D).
4. Especificación YACC y aquellos módulos adicionales que se hayan codificado para la construcción del analizador sintáctico.

3.4 Prueba de la Práctica

Será necesario confeccionar un programa de ejemplo en el lenguaje asignado donde aparezcan:

- **Funciones/Procedimientos** anidados a tres niveles y en paralelo ({A{B{C}}D{E}}), con diferencias en cuanto a número y tipo de los argumentos, así como las llamadas a estas funciones/procedimientos declarados. Esto no debe poner de manifiesto el ámbito de uso de los subprogramas anidados.
- **Sentencias tipo del lenguaje asignado** en al menos dos lugares y sentencias compuestas con más de una sentencia (ejemplo: sentencia condicional donde la condición conlleva una expresión compuesta).
- **Expresiones aritmético-lógicas** complejas.
- **Expresiones del tipo estructurado** donde se pueda ver que es posible operar con el tipo estructurado asignado al lenguaje y con constantes o variables de tipo básico que sea compatible.
- **Variables declaradas** de cada tipo básico y cuatro variables del tipo estructurado de cada tipo básico. Dichas variables podrán ser usadas por el profesor para introducir comprobaciones de tipo en las expresiones.

Un ejemplo de código para la prueba de la práctica puede verse en el **apéndice E** (ver página 49).

Prueba del Sintáctico para la defensa de la práctica

1. Sobre la especificación YACC se le aplicará el YACC/BYACC/BISON para comprobar que sólo detecta el conflicto desplaza/reduce (shift/reduce) correspondiente a la especificación de la sentencia if-then-else.
2. Sobre el programa de ejemplo confeccionado según los requisitos anteriores (libre de errores), se le pasa el analizador sintáctico para comprobar que no detecte errores.
3. Una vez superadas las pruebas anteriores, el profesor introducirá los siguientes tipos de errores que se muestran así como la puntuación obtenida si se detectan:

Puntuación	Error introducido
0.1	**Error léxico en las declaraciones**, debiendo detectar el error léxico y, en su caso, el error sintáctico derivado.
0.1	Se **omite una ","** en una línea de **declaración de variables** con, al menos, tres variables y también se omite el ";"de dicha línea. En la **siguiente línea de declaración** se **omite otra ","**. Deberá detectar un error en cada línea. En la primera línea debe haber tres variables como mínimo para que se pueda detectar el error en ambas líneas. Ello se debe a la naturaleza de acción tanto de Lex y Yacc en el que contabilizan un símbolo de anticipación. De esa forma, si en la primera línea hay sólo dos variables, no se detectaría el error introducido en la línea siguiente.
0.1	Error en los argumentos de la **cabecera de un subprograma**.
0.3	Error en **expresiones aritméticas complejas**.
0.4	Error en **expresiones aritmético-lógicas de sentencias compuestas**, debiendo detectar todos los errores introducidos.

La puntuación se obtendrá como suma de los puntos asignados para cada tipo de error introducido y detectado con éxito.

En el **apéndice C** (ver página 37) se muestra información de cierto detalle sobre la utilización de YACC, mientras que en el **apéndice D** (ver página 43) se muestra información adicional sobre el estilo de las producciones y tratamiento de los errores en cuanto a informar mediante un mensaje certero el error que se produce y cómo poder recuperarnos ante un error.

Práctica 4

Analizador Semántico

4.1 Objetivos

El objetivo de la práctica es construir el analizador semántico para el lenguaje diseñando. Para simplificar y estructurar esta práctica, se dividirá en dos partes:

1. Implementación de la Tabla de Símbolos.
2. Implementación de las Acciones Semánticas.

4.2 Partes de la Práctica

4.2.1 Implementación de la Tabla de Símbolos

En esta parte de la práctica se introducirán las acciones YACC necesarias para construir y visualizar la tabla de símbolos. La visualización debe presentar la actualización de la tabla de símbolos a medida que se analiza cada bloque. Dichas acciones se incluirán junto a las producciones que definan las declaraciones. Esta primera parte sólo será de utilidad para la puesta a punto de la práctica completa y, por lo tanto, será de gran utilidad para los alumnos.

La única comprobación semántica que se puede realizar, en esta parte, es detectar si una variable ha sido declarada, en un mismo bloque, más de una vez.

También es posible usar el paquete de funciones de Sefasgen (ver **apéndice H** en página 63 para mayor información) para implementar el apartado de la tabla de símbolos y poder realizar pruebas paso a paso de cómo evoluciona la misma mientras se compila un texto fuente.

Las tareas a realizar en esta parte de la práctica son:

1. Determinar cuales son todas las acciones semánticas necesarias para poder realizar las comprobaciones y en qué reglas de la gramática abstracta deben ser insertadas esas comprobaciones.
2. Una vez obtenido lo anterior, diseñar las estructuras de datos (ver **apéndice F** en página 51) y las operaciones que vamos a realizar sobre dichas estructuras de datos, con el fin de realizar las acciones semánticas establecidas.

3. Insertar en la especificación YACC las acciones necesarias para construir y visualizar la tabla de símbolos, así como realizar la comprobación semántica propia de esta parte.

Pruebas para la Tabla de Símbolos

Se sugiere que cada grupo de prácticas pruebe esta primera parte de la practica 4 usando el programa prueba confeccionado para la prueba de la práctica anterior, en donde podrá comprobar el comportamiento de la tabla de símbolos. También se deberían introducir errores en declaración de variables y se comprobarán que los mensajes de error son los adecuados.

4.2.2 Comprobaciones Semánticas

En esta parte de la práctica se introducirán las acciones YACC necesarias para el resto de las comprobaciones semánticas. Se debe extender la especificación YACC obtenida en la práctica anterior para realizar las comprobaciones semánticas propias de cada lenguaje. Para ello, se añadirán el código necesario para manejar las estructuras de datos diseñadas en el apartado anterior y realizar las acciones semánticas necesarias. El programa ejecutable obtenido debe realizar todo el análisis semántico y sintáctico en una sola pasada.

Hay que tener en cuenta que, según las opciones escogidas por cada grupo de prácticas, algunas de las comprobaciones, acciones o elementos de la estructura de datos no van a ser necesarias. Cada grupo debe razonar cuáles son los elementos que se necesitan y cuáles no.

Por ámbito de un identificador (variables, parámetros, funciones y procedimientos) entendemos el bloque en el que está declarado y todos aquellos bloques incluidos en él, directa o indirectamente. En el caso de parámetros de procedimientos y funciones, entendemos que el bloque en el que están declarados es el que forma el cuerpo del subprograma. En adelante, supondremos que cada expresión, sub-expresión o término que aparece en el programa tiene un tipo determinado de entre los permitidos. El lenguaje realizará comprobación fuerte de tipos, al estilo de Pascal (no se admiten conversiones de tipo).

Las comprobaciones semánticas que debe realizar nuestro compilador van a ser las siguientes:

- El punto en que se usa un identificador pertenece a su ámbito.
- En el ámbito de un identificador puede declararse otro con el mismo nombre, pero no en el mismo bloque en el que está declarado el primero.
- En el caso de asignaciones, el tipo de la parte izquierda debe coincidir con el tipo de la expresión en la parte derecha.
- En el caso de llamadas a procedimientos y funciones, el tipo, número y orden de las expresiones que forman los parámetros actuales deben coincidir con el tipo, número y orden de los parámetros formales especificados en su declaración.
- En el caso de expresiones que incluyan un operador (ya sean unarios, binarios o ternarios[1], comprobar que el operador es compatible con el tipo de las sub-expresiones sobre las que actúa (ver **apéndice A**).

[1] En algún caso, para el tipo estructurado, es necesario definir una operación concreta empleando tres operandos y dos operadores, por lo tanto, ternarios

4.3 Prueba de la Práctica

Se partirá del programa confeccionado para la defensa de la práctica 3 sin errores y, a partir de ahí, el proceso de defensa y corrección de la práctica será:

1. Se pasará el analizador sobre el programa prueba (sin errores) comprobando que está libre de errores.
2. El profesor introducirá los siguientes errores, viendo en la siguiente tabla las calificaciones parciales de cada apartado.

Puntuación	Error introducido
0.15	**Error léxico en un tipo** de la declaración de variables, generando error léxico, sintáctico y semántico.
0.2	En el **cuerpo de un subprograma** se **declara** uno de sus **argumentos**. Debe indicar que no es posible.
0.5	**Llamadas a subprogramas** con **número y tipo** de los **argumentos** erróneos y en **ámbitos fuera** de su **alcance**. Debe informar de estos tres tipos de errores semánticos.
0.2	Error de **tipo en expresiones lógicas y aritméticas**.
0.2	Error de **tipo en expresiones compuestas** del tipo de dato estructurado que se le haya asignado (lista, pila, conjunto, array).

NO SE DEBE PRESENTAR LA TABLA DE SÍMBOLOS. SÓLO LOS ERRORES LÉXICOS, SINTÁCTICOS Y SEMÁNTICOS.

La puntuación se obtendrá como suma de los puntos asignados para cada tipo de error introducido que hayan sido detectados.

4.4 Documentación a Presentar

1. Documentación de la primera práctica.
2. Tabla de tokens (incluyendo atributos, en su caso).
3. Listado del código fuente correspondiente a la implementación del analizador semántico. Esto incluye tanto las acciones incluidas en la especificación YACC como las implementaciones de dichas acciones en las rutinas de usuario.

Finalmente, en el **apéndice F** (ver página 51) se encuentra información útil para la realización de las comprobaciones semánticas y para la definición de las estructuras de datos.

Práctica 5

Generación de Código Intermedio

5.1 Objetivos

Obtener el generador de código intermedio. Dicho código intermedio se basa en un lenguaje C pero con expresiones simples con objeto de poder probar la correcta ejecución del código generado.

5.2 Puntuación de la Práctica

Parte	Puntuación	Realización
Obligatoria	**1.25**	Generación de **código intermedio elemental**, sin incluir la generación de código para subprogramas con anidamiento ni para la estructura de datos especificada en el lenguaje fuente.
Optativa	**0.5**	Generación de código intermedio elemental incluyendo el código correspondiente a **subprogramas anidados**.
Optativa	**1.0**	Generación de código intermedio elemental incluyendo la **definición y gestión del tipo estructurado** especificado en el lenguaje fuente (lista, pila, conjunto, array). Se debe elaborar un programa de ejemplo que manipule datos de este tipo y que ofrezca los resultados correctos.

5.3 Lenguaje Intermedio

Será un lenguaje C cuyas instrucciones mantienen las restricciones de las de un lenguaje en cuartetos (excepto en la llamadas a subprogramas, el uso de subprogramas de entrada y salida y algunas estructuras de datos). Las instrucciones que pueden aparecer en el lenguaje intermedio se enumeran a continuación:

1. Podrá incluir las librerías que crea oportunas.
2. Declaraciones de tipos elementales C.
3. Declaración de funciones.
4. Declaración de array, cadenas, y cualquier otra estructura que crea conveniente para implementar la estructura de datos especificada en el lenguaje fuente.

5. Se mantiene la estructura de bloques.
6. Las expresiones son de la forma:

```
<Operando> <OpBinario> <Operando>
<OpUnario> <Operando>
<Operando>
```

   donde `<Operando>` es un valor de tipo elemental, o un identificador (si se tienen como estructura los array, el identificador puede ser el elemento de un array), `<OpBinario>` y `<OpUnario>` son cualquier operador binario y unario respectivamente.
7. Asignación: `(<Ident>|<ElemArray>) = (<Expresion>|<LlamFuncion>)`.
8. Salto incondicional: `goto <etiqueta>`.
9. Sentencias de lectura y escritura: Se utilizarán las funciones de C ya conocidas `scanf` y `printf`.
10. Llamada a subprograma: `<nombre_subprog> ( <lista_argumentos> )`.

5.4 Documentación a Presentar

1. Documentación de la primera práctica.
2. Tabla de tokens (incluyendo atributos, en su caso).
3. Especificación LEX y YACC y ficheros fuentes C que se usen en esta etapa.

5.5 Prueba de la Práctica

Los alumnos prepararán dos o más ejemplos en su lenguaje, de forma que permitan comprobar todas las características del lenguaje que se ha implantado. Como referencia, pueden usarse estos dos ejemplos. Ambos realizan la descomposición de un número entero en sus factores primos.

Listing 5.1: gencodigo1.c

```
#include <stdio.h>

/** escribe la descomposición de un numero entero en sus factores primos,
*** usa exclusivamente: multiplicacion, division y suma de enteros
**/
int main(int argc, char * argv[] )
{
   int n, curr ;
   printf("introduce␣numero␣:␣");
   scanf("%d",&n);
   printf("␣%d␣==␣",n);
   curr = 2 ;
   while( curr <= n )
   {
        int d = n/curr ;

        if ( d*curr == n )  /* curr divide a n */
        {
            printf("*␣%d␣",curr);
            n = n/curr ;
        }
```

```
        else
           curr = curr+1 ;
   }
   printf("\n");
}
```

Listing 5.2: gencodigo2.c

```
#include <stdio.h>

/** escribe la descomposición de un numero entero en sus factores primos,
*** usa exclusivamente: multiplicacion, division y suma de enteros
**/
int main(int argc, char * argv[] )
{
   int n, curr = 2, ultim = 0, cuenta = 0, primero = 1 ;
   printf("introduce␣numero␣:␣");
   scanf("%d",&n);
   printf("%d␣␣==␣",n);
   curr = 2 ;
   while( curr <= n )
   {
        int d = n/curr ;

        if ( d*curr == n )  /* curr divide a n */
        {
            if ( curr != ultim )
            {
               ultim = curr ;
               cuenta = 1 ;
            }
            else
               cuenta = cuenta + 1 ;

            n = n/curr ;
        }
        else /* curr no divide a 'n' */
        {
            if ( cuenta > 0 )
            {
               if ( primero == 0 ) printf("␣*");
               primero = 0 ;
               printf("␣%d",curr) ;
               if ( cuenta > 1 ) printf("^%d",cuenta) ;
            }

           curr = curr+1 ;
           cuenta = 0 ;
        }
   }
   if ( cuenta > 0 )
   {
      if ( primero == 0 ) printf("␣*");
      primero = 0 ;
      printf("␣%d",curr) ;
      if ( cuenta > 1 ) printf("^%d",cuenta) ;
   }

   printf("\n");
}
```

Para cada ejemplo se obtendrá un fichero (que será visible para su comprobación) con su traducción al lenguaje intermedio, este fichero se compilará en C usando el compilador GCC o cualquier otro que admita ANSI C (en ningún caso se usará g++) y se probará. Si todos los ejemplos funcionan se obtiene la nota correspondiente, en caso contrario la práctica no puntuará.

5.6 Estructura del Programa en Lenguaje Intermedio

La estructura de un programa en lenguaje intermedio generador por el traductor tendrá la siguiente estructura:

- Inclusión de librerías C que se estime oportunas.
- Inclusión del fichero con las funciones y, si es necesario, la declaración de tipos para la gestión del tipo estructurado que haya sido asignado al lenguaje en la forma: `#include "dec_dat"`
- Inclusión del fichero con la declaración de funciones: `#include "dec_fun"`
- Código generado. Hay que tener en cuenta que las variables del procedimiento `main` deben declararse como globales, es decir, se declaran antes de `int main (...)`.

El apartado segundo sólo será necesario si opta por la generación de código para la estructura de datos especificada en el lenguaje fuente. El apartado tercero sólo será necesario si opta por la generación de código para los subprogramas.

5.7 Traducción de las Expresiones complejas a Expresiones Simples

5.7.1 Nombres de Variables Temporales

Para generar código necesitará una función que genere nombres de variables temporales y obtenga su declaración. La función `char *temporal(void) ;` debe generar el nombre de una variable temporal como `tempN`, donde `N` es un número entero. Cada vez que obtenga una variable temporal deberá declararla en el fichero objeto.

Ejemplo 1 *En la siguiente tabla se muestra una sentencia en lenguaje fuente tipo Pascal y su traducción a lenguaje intermedio (las variables `a,b,c` son de tipo entero y `d,h` son booleanas.*

`h := (a+b-c > c-3) and not d`

`int temp0 ;`
`temp0= a+b ;`
`int temp1 ;`
`temp1= temp0-c ;`
`int temp2 ;`
`temp2= c-3 ;`
`int temp3 ;`
`temp3= temp1 > temp2 ;`
`int temp4 ;`
`temp4= !d ;`
`int temp5 ;`
`temp5= temp3 && temp4 ;`
`h= temp5 ;`

Cuando se traduzca una instrucción se abrirá un bloque al comienzo de la traducción y se cerrará el bloque al final de la traducción. Esto permite optimizar la memoria requerida en ejecución por las variables temporales introducidas, ya que se liberaría la memoria de las variables temporales una vez finalizado el bloque.

Ejemplo 2 *Dadas dos sentencias que manejan expresiones compuestas en lenguaje fuente, se muestra en la siguiente tabla la traducción empleando estructura de bloques para cada sentencia.*

```
a := b+c-f(d) ;
m := j*4-a ;
```

```
{  // comienzo de la traducción de la primera asignación
   int temp0 ;
   temp0= b+c ;

   int temp1 ;
   temp1= f(d) ;

   int temp2 ;
   temp2= temp0-temp1 ;

   a= temp2 ;
}  // fin de la primera asignación.

{  // comienzo de la traducción de la segunda asignación
   int temp3 ;
   temp3= j*4 ;

   int temp4 ;
   temp4= temp3-a ;

   m= temp4 ;
}  // fin de la segunda asignación.
```

5.7.2 Abributo Sintetizado "Nombre"para las Expresiones

Con el fin de generar código, las expresiones deben sintetizar como atributo el nombre de la variable temporal que se ha generado para almacenar el resultado y además el descriptor de tipo para, en caso de expresiones complejas, se pueda realizar la comprobación de tipos en todo momento. Será necesario ampliar la estructura de los atributos de los símbolos para contemplar el código generado. Para ello bastará con añadir un campo de tipo cadena de caracteres en el que se irá concatenando el código.

5.8 Traducción de las Instrucciones de Control de Flujo

Para traducir las instrucciones de control de flujo necesitamos usar etiquetas, para obtenerlas se utilizará una función `char *etiqueta(void)`, que generará nombres para las etiquetas de la forma `etiquetaN`, donde `N` es un número entero. Cada instrucción de control de flujo necesita unas etiquetas ya sean las de entrada y salida en el caso de ciclos, o la de salida, o la parte `else` en el caso del `if`. Para almacenar las etiquetas requeridas por las instrucciones de control no es suficiente el uso de variables globales, ya

que estas instrucciones pueden aparecer anidadas. Una forma de solucionar este problema consistiría en utilizar la tabla de símbolos para almacenar las etiquetas. Para ello, debemos añadir el descriptor siguiente:

```
typedef struct {
   char *EtiquetaEntrada ;
   char *EtiquetaSalida ;
   char *EtiquetaElse ;
   char *NombreVarControl ;
} DescriptorDeInstrControl ;
```

donde:

`EtiquetaEntrada` contendrá el nombre de la etiqueta de entrada.

`EtiquetaSalida` contendrá la etiqueta de salida.

`EtiquetaElse` en el caso de `if`, contendrá el nombre de la etiqueta del `else`.

`NombreVarControl` guardará el nombre de la variable de control para bucles definidos o estructura case. Su gestión se muestra con mayor detalle en las siguientes secciones.

5.8.1 Sentencia Condicional IF-THEN-ELSE

La traducción de una sentencia `if` producirá las siguientes acciones:

1. Se genera la etiqueta de salida y la del `else`.
2. Se insertará en la tabla de símbolos el correspondiente descriptor de instrucción de control, con la etiqueta `else` y la de salida.
3. Se procesa la expresión que forma la condición, de forma que esté representada por una variable temporal.
4. Se emite la sentencia de salto condicional hacia el `else`.
5. Se procesan las instrucciones correspondientes a la parte `if`.
6. Se emite el salto incondicional a la etiqueta de salida.
7. Se emite la etiqueta de la parte `else`.
8. Si existe una parte `else`, se emiten las instrucciones correspondientes.
9. Se emite la etiqueta de salida.
10. Se eliminan en la tabla de símbolos las entradas hasta el descriptor de instrucción de control.

Ejemplo 3 *Dadas las siguientes sentencias en lenguaje fuente según sintaxis Pascal, se ofrece la traducción a cuartetos empleando C.*

```
if (a>b) then
 begin
 a := b+c-f(d) ;
 m := j*4-a
 end
else
 h := 3 ;
```

```
{  // comienzo de la traducción del if
   int temp0 ;
   temp0= a>b ;
   if (!temp0) goto etiqueta1 ;
   {  // traducción del begin
      {  // comienzo de la traducción de la asignación
         int temp1 ;
         temp1= b+c ;

         int temp2 ;
         temp2= f(d) ;

         int temp3 ;
         temp3= temp1-temp2 ;

         a= temp3 ;
      }  // fin de la traducción de la asignación

      {  // comienzo de la traducción de la asignación
         int temp4 ;
         temp4= j*4 ;

         int temp5 ;
         temp5= temp4-a ;

         m= temp5 ;
      }  // fin de la traducción de la asignación
   }  // traducción del end

   goto etiqueta2 ;

etiqueta1:
   {  // comienzo de la traducción de la asignación
      h= 3 ;
   }  // fin de la traducción de la asignación
}  // fin de la traducción del if

etiqueta2:
```

5.8.2 Bucles Indefinidos

Se presentan las acciones a realizar cuando se traduce un bucle `while`, en el caso de otros bucles indefinidos necesitaremos las etiquetas de entrada y de salida, y solo realizar modificaciones sobre el siguiente esquema:

1. Se genera la etiqueta de entrada y la de salida.

2. Se inserta en la tabla de símbolos el correspondiente descriptor de instrucción de control, con las etiquetas generadas.
3. Se sitúa en el código objeto la etiqueta de entrada.
4. Se procesa la expresión que forma el la condición, de forma, que esté representada por una variable temporal.
5. Se emite la sentencia de salto condicional hacía la etiqueta de salida.
6. Se procesan las instrucciones al cuerpo del bucle.
7. Se emite el salto incondicional a la etiqueta de entrada.
8. Se emite la etiqueta de salida.
9. Se eliminan en la tabla de símbolos las entradas hasta el descriptor de instrucción de control.

5.8.3 Bucles Definidos

Básicamente, un bucle definido contiene una variable de control, una expresión inicial, una expresión final y las sentencias que constituyen el cuerpo del bucle. En el esquema, que se presenta a continuación, tenemos la implementación del bucle suponiendo un incremento de 1 para la variable de control del bucle.

1. Generar la etiqueta de entrada y la de salida.
2. Insertar en la tabla de símbolos el correspondiente descriptor de instrucción de control con la variable de control de las etiquetas obtenidas.
3. Traducir la expresión inicial.
4. Emitir la asignación de la expresión inicial a la variable de control.
5. Situar en el código objeto la etiqueta de entrada.
6. Traducir la expresión final.
7. Emitir las sentencias que comparan la expresión final con la variable de control y la asignan a una variable temporal.
8. Emitir la sentencia de salto condicional, que depende la variable temporal del apartado anterior, hacia la etiqueta de salida.
9. Procesar las instrucciones en cuerpo del bucle.
10. Emitir los cuartetos que incrementa la variable de control.
11. Emitir el cuarteto que representa el salto incondicional a la etiqueta de entrada.
12. Situar la etiqueta de salida.
13. Eliminar de la tabla de símbolos las entradas hasta el descriptor de instrucción de control.

5.8.4 Sentencia CASE

En la sentencia case tenemos una expresión que da valor al discriminante y una serie de pares compuestos por una expresión y un bloque de sentencias. Su traducción se puede realizar de la siguiente manera:

1. Traducir la expresión del discriminante a cuartetos y almacenar su valor en una variable temporal.
2. Insertar en la tabla de símbolos el correspondiente descriptor de instrucción de control con la variable de control (variable temporal que contiene el valor del discriminante).
3. Para cada par expresión-bloque de sentencias, las acciones a realizar son:
   (a) Se obtiene la etiqueta de salida.
   (b) Se inserta en la tabla de símbolos un descriptor de instrucción de control, asociado con el bloque de sentencias, en el cuál, sólo necesitamos la etiqueta la de salida.
   (c) Se traduce la expresión guardando su valor en una variable temporal.
   (d) Se emiten los cuartetos que comparan el valor de la expresión (variable temporal calculada en el apartado anterior) con el valor del discriminante (variable de control que está situada en el descriptor de control asociado al case) y se asigna a una variable temporal.
   (e) Se emite la sentencia de salto condicional, que depende la variable temporal del apartado anterior, hacia la etiqueta de salida.
   (f) Se procesan las instrucciones del bloque de sentencias.
   (g) Se emite la etiqueta de salida.
   (h) Se eliminan de la tabla de símbolos las entradas hasta el descriptor de instrucción de control asociado al bloque sentencias.
4. Eliminar en la tabla de símbolos las entradas hasta el descriptor de instrucción de control.

5.9 Traducción de las Estructuras de Datos

1. **Listas**
   - Implementar las funciones que realicen las operaciones de las listas mediante funciones C que deben ser incluidas en el fichero `dec_dat`.
   - Traducir todas las operaciones de las listas por las llamadas a sus funciones correspondientes.
   - Definir el tipo que se va implementar la listas e incluirlo en `dec_dat`.
   - Las declaraciones de las listas en el programa fuente se traducirán a declaraciones de tipo definido en `dec_dat`.
2. **Arrays**
   - Implementar las funciones que realicen las operaciones suma, resta, multiplicación y división de array elemento a elemento, mediante funciones C e incluirlas en el fichero `dec_dat`.
   - Las declaraciones de array en el programa fuente se traducen a declaraciones de array en C.
   - Traducir todas las operaciones de array que existan en el programa fuente por las llamadas a funciones correspondientes.

3. **Conjuntos**
   - Se implantará mediante un lista de elementos.
   - Definir el tipo que se va a utilizar para implementar los conjuntos e incluirlo en `dec_dat`.
   - Implementar las funciones que realice las operaciones unión e intersección de conjuntos, y la asignación de valores a un conjunto mediante funciones C, e incluirlas en el fichero `dec_dat`.
   - Traducir Las declaraciones de conjuntos en el programa fuente a declaraciones del tipo incluido en `dec_dat`.
   - Las operaciones de conjuntos que existen en el programa fuente se traducen por las llamadas a funciones adecuadas contenidas en `dec_dat`.
4. **Pilas**
   - Definir el tipo que se va a utilizar para implementar las pilas e incluirlo en `dec_dat`.
   - Implementar las funciones que realicen las operaciones `push`, `pop` y `consulta` del tope de la pila mediante funciones C, e incluirlas en el fichero `dec_dat`.
   - Las declaraciones de pilas en el programa fuente se traducen a declaraciones del tipo definido en `dec_dat`.
   - Traducir todas las operaciones que existan en el programa fuente por las llamadas a las funciones implantadas para tal fin.

5.10 Traducción de los Subprogramas

La declaración y traducción de las todos los subprogramas del texto fuente deben estar en el fichero `dec_fun`. Las acciones a realizar al detectar un subprograma son las siguientes:

1. Cambiar del fichero donde se está generando código al fichero `dec_fun`.
2. Declarar y traducir el subprograma de acuerdo al lenguaje intermedio.
3. Al finalizar el subprograma volver al fichero inicial para seguir la traducción del resto del código.

No es necesario tener consideraciones especiales para el caso de subprogramas anidados dado que el compilador de C de GNU soporta funciones anidadas, por lo tanto, se traducen tal cual.

Apéndice A

Estructuras de Datos Consideradas como Tipo Elemental

En este apéndice se muestra la descripción completa a nivel de sintaxis y de semántica de los tipos estructurados que se incorporarán como tipos elementales para la construcción del traductor del lenguaje de programación orientado al problema.

Se considera que, salvo que devuelva valores de tipo base no estructurado, lo que devuelve es siempre una copia del dato tras efectuar la operación. Ésto se considera esencial para evitar problemas laterales muy típicos en determinados lenguajes de programación con C y C++.

A.1 Listas Dinámicas

Una lista es una estructura de datos compuesta que permite almacenar una colección dinámica de valores en un orden determinado y en el que se dispone de un cursor que señala a uno de los elementos. Dicho valor es representado mediante un dato de tipo entero.

- *Declaración*: La lista puede ser declarada de cualquier tipo básico (entero, real, carácter o booleano). Puesto que se trata de listas dinámicas, en la declaración no se especifica el tamaño ya que vendrá determinado por el número de elementos que tenga alojados en cada momento.
- *Ejemplos de declaración*: Dado un lenguaje con palabras reservadas en inglés.

```
/* Lenguaje tipo C */
       list of int   lx, lz;
       list of char  ly, la ;
```

```
/* Lenguaje tipo Pascal */
       c1, c2 : list of integer ;
       c3, c4 : list of boolean ;
```

- *Agregados*: Un agregado de tipo lista se especifica mediante una secuencia de expresiones del mismo tipo base.

```
[1, 43, 5, 0, 0]                /** lista de enteros **/
['z', 'A', 'B']                 /** lista de caracteres **/
[2.5, 4.9*c, 3.0+f(b,c), 0.0]   /** lista de reales **/
[true, false, true, true]       /** lista de booleanos **/
```

- *Sentencias*: Aparecen estas nuevas sentencias para el tratamiento de listas:

Nombre	Lexema	Argumento	Ejemplo	Resultado
avanzar	$>>$	lista	$l>>$	Avanza el cursor en una posición
retroceder	$<<$	lista	$l<<$	Retrocede el cursor en una posición
cursor al comienzo de la lista	\$	lista	$\$l$	Lleva el cursor al comienzo de la lista

- *Operadores*: Se definen los siguientes operadores, teniendo en cuenta que son formas de expresión y no de sentencia, salvo que se indique como *sentencia*, es decir, se devuelve un nuevo elemento del tipo base de la lista o una nueva lista modificada tras realizar la operación correspondiente.

Nombre	Tipo	Lexema	Operandos	Ejemplo	Resultado
longitud	unario	#	lista	$\#l$	Devuelve el número de elementos de l
elemento actual	unario	?	lista	$?l$	Devuelve el elemento actual de la lista
elemento posición	binario	@	lista y valor	$l@x$	Devuelve el elemento de la posición x
añadir elemento en una posición	ternario	$++$ y @	lista, valor y posición	$l++x@z$	Devuelve una copia de l con x añadido en la posición z
borrar elemento en una posición	binario	$--$	lista y posicion	$l--x$	Devuelve una copia de l con el elemento en la posición x borrado
borrar lista a partir de una posición	binario	%	lista y posición	$l\%x$	Devuelve una copia de l sin los elementos a partir de la posición x
concatenar listas	binario	$**$	dos listas	$l_1 ** l_2$	Añade los elementos de l_2 en l_1 y devuelve una copia
suma	binario	$+$	lista y valor	$l+x$	suma de x con cada elemento
			valor y lista	$x+l$	suma de cada elemento con x
resta	binario	$-$	lista y valor	$l-x$	resta de x con cada elemento
producto	binario	$*$	lista y valor	$l*x$	producto de x con cada elemento
			valor y lista	$x*l$	producto de cada elemento con x
división	binario	$/$	lista y valor	l/x	división de x con cada elemento

- *Reglas semánticas*: Tal y como muestra la tabla anterior, los operadores unarios tienen como argumento una lista de cualquier tipo base. Los operadores binarios manejan la lista de tipo base y, bien un elemento que debe ser del mismo tipo base o bien la posición que debe ser de tipo entero o ambos (operador ternario).

  Existen tres operadores unarios para la realización del recorrido de una lista. Dos de ellos hacen desplazamientos, uno de retroceso y otro de avance y el tercero sitúa el cursor al comienzo de la lista.

  Por último, la concatenación de listas debe actuar sobre listas que posean el mismo tipo base con independencia del número de elementos que en ellas se alojen.

A.2 Arrays 1D y 2D

Este tipo sirve para declarar variables o expresiones que contienen arrays de expresiones de un mismo tipo básico. En concreto, permitir declarar array de enteros, reales, booleanos o caracteres. El tipo de los elementos se le denomina *tipo base* del tipo array.

- *Declaración*: Debe especificarse el tipo base, las dimensiones (una o dos como máximo) y el tamaño entero de cada dimensión indicado mediante una constante. El rango de los índices de cada dimensión será para todos los casos desde 1 hasta el valor del tamaño en el caso de Pascal y desde 0 hasta el tamaño menos uno, para el caso de C.

- *Ejemplo de declaración*: Dado un lenguaje con palabras reservadas en inglés.

```
/* Lenguaje tipo C */
      int a[20], b[10,3];
      char c[10], h[2,4];
```

```
/* Lenguaje tipo Pascal */
      c1, c2 : array of integer[20] ;
      v1, v2 : array of character[20,3] ;
```

- *Agregados*: Los agregados del tipo array se especifican como una lista de expresiones de un mismo tipo base.

- *Ejemplos de agregados*:

```
{1, 43*(a+5), 5, 0, 0}      /** array 1D de enteros de tamaño 5 **/
{'z', 'A', 'B'}             /** array 1D de caracteres de tamaño 3 **/
{2.5, 4.9+b ; 3.0, 0.0}     /** array 2D de reales de tamaño 2x2 **/
```

- *Operadores*: Se definen estos cinco operadores, teniendo en cuenta que definen forma de expresión y no forma de sentencia, es decir, se devuelve un nuevo valor del tipo base del array o un nuevo array tras realizar la operación.

Nombre	Tipo	Lexema	Operandos	Ejemplo	Resultado
suma	binario	$+$	dos arrays	$a_1 + a_2$	suma elemento a elemento
			array y valor	$a + x$	suma de x con cada elemento
			valor y array	$x + a$	suma de cada elemento con x
resta	binario	$-$	dos arrays	$a_1 - a_2$	resta elemento a elemento
			array y valor	$a - x$	resta de cada elemento con x
producto	binario	$*$	dos arrays	$a_1 * a_2$	producto elemento a elemento
			array y valor	$a * x$	producto de cada elemento con x
			valor y array	$x * a$	producto de x con cada elemento
división	binario	$/$	dos arrays	a_1/a_2	división elemento a elemento
			array y valor	a/x	división de cada elemento con x
multipl.	binario	$**$	dos arrays	$a_1 * *a_2$	multiplicación de matrices

Existen dos formas de multiplicar arrays. La primera toma dos operandos de igual dimensión y tamaño y realiza el producto elemento a elemento. La segunda forma es mediante la multiplicación de arrays donde el número de columnas del primer operando debe coincidir con el número de filas del segundo. El resultado en ese caso será un nuevo array con el mismo número de columnas que el primer operando y número de filas del segundo operando.

- *Acceso a los elementos*: Se realizará mediante índices enteros para cada dimensión separados por coma y entre paréntesis (ejemplo, $a[2,3]$ nos indica el elemento situado en la segunda fila y en la tercera columna).

- *Reglas semánticas*: La asignación requiere que ambas partes sean arrays que coincidan tanto en el tipo base, dimensión y tamaño.

A.3 Conjuntos

Este tipo sirve para declarar variables (o escribir expresiones) que contienen (o devuelven) colecciones homogeneas de un numero finito de valores no repetidos de un tipo primitivo (no estructurado). Ese tipo se denomina su 'tipo base', y se contempla el conjunto vacío (que no tiene tipo base).

- *Declaración*: Debe especificarse el tipo base de la variable que se declara.
- *Ejemplos de declaración*: Dado un lenguaje con palabras reservadas en inglés.

```
/* Lenguaje tipo C */
       set int  c1, c2 ;
       set char letras ;
       set float s ;
```

```
/* Lenguaje tipo Pascal */
       c1, c2 : set of integer ;
       letras : set of character ;
            c : set of integer ;
```

- *Agregados*: Los agregados del tipo se especifican como una lista (posiblemente vacía) de constantes de un tipo base (un tipo primitivo). La lista vacía denota el conjunto vacío. Si la constante incluye expresiones de tipo T, entonces la expresión que forman es de tipo 'conjunto de T', si la constante denota el conjunto vacío, el tipo es 'conjunto' (no tiene asociado tipo base).
- *Ejemplos de agregados*[1]:

```
{1, 43, 5/a}          /** tipo == conjunto de enteros **/
{'z', 'A', 'B'}       /** tipo == conjunto de caracteres **/
{}                    /** tipo == conjunto **/
{'z', 3.4*f(x)}       /** error: elementos de tipo distinto **/

s, c : set of integer ;

s:= { c } ;      /** error: elemento de tipo no primitivo **/
```

- *Operadores*: Se definen los siguientes operadores, teniendo en cuenta que son formas de expresión y no de sentencia, es decir, se devuelve un nuevo elemento del tipo base del conjunto, un valor lógico o un nuevo conjunto tras realizar la operación correspondiente.

Nombre	Tipo	Lexema	Operandos	Ejemplo	Conjunto o valor
unión	binario	$+$	dos conjuntos	$c_1 + c_2$	valores en c_1 o en c_2
intersección	binario	$*$	dos conjuntos	$c_1 * c_2$	valores en c_1 y en c_2
diferencia	binario	$-$	dos conjuntos	$c_1 - c_2$	valores en c_1 y no en c_2
inclusión estricta	binario	$<$	dos conjuntos	$c_1 < c_2$	'true' si $c_1 \subset c_2$
inclusión	binario	$<=$	dos conjuntos	$c_1 <= c_2$	'true' si $c_1 \subseteq c_2$
superconjunto estricto	binario	$>$	dos conjuntos	$c_1 > c_2$	'true' si $c_1 \supset c_2$
superconjunto	binario	$>=$	dos conjuntos	$c_1 >= c_2$	'true' si $c_1 \supseteq c_2$
pertenencia	binario	*in*	valor y conjunto	x *in* c	'true' si $x \in c$
suma	binario	$+$	conjunto y valor	$c + x$	suma a todos los elementos x
			valor y conjunto	$x + c$	y devuelve una copia del conjunto
resta	binario	$-$	conjunto y valor	$c - x$	resta a todos los elementos x y devuelve una copia del conjunto
producto	binario	$*$	conjunto y valor	$c * x$	multiplica x a todos los elementos y
			valor y conjunto	$x * c$	devuelve una copia del conjunto
división	binario	$/$	conjunto y valor	c/x	divide x a todos los elementos y devuelve una copia del conjunto

[1] En lenguajes tipo Pascal, donde los comentarios al código se realiza usando delimitadores basados en llaves, se puede optar por modificar estos delimitadores por otros (por ejemplo, usar la misma forma que en lenguaje C) o emplear otros delimitadores para considerar agregados de tipo conjunto.

- *Reglas semánticas*:
  - Las expresiones con operadores binarios aplicados a dos conjuntos son correctas si y solo si se cumple alguna de estas dos condiciones sobre el tipo de las dos subexpresiones que la forman:
    - Una es de tipo 'conjunto de T' y la otra es de ese mismo tipo o de tipo 'conjunto' (la expresión completa es de tipo 'conjunto de T' para los operadores $+, *, -$ y de tipo lógico para el resto).
    - Ambas son de tipo 'conjunto' (la expresión completa es de tipo 'conjunto' para los operadores $+, *, -$ y de tipo lógico para el resto).
  - En los casos en los que se apliquen los operadores aritméticos $+, -, *, /$ entre expresiones de tipo T y un conjunto, la expresión es correcta si y solo si T es entero o flotante y el conjunto es de tipo 'conjunto de T' o 'conjunto' (en ese caso, la expresión que resulta será de tipo 'conjunto de T', siendo T el tipo del operando que puede ser, como se menciona anteriormente, entero o flotante).
  - La expresión *x* in *s* es correcta (y de tipo lógico) si y solo si *x* es una expresión de un tipo primitivo T y *s* es una expresión de tipo 'conjunto de T' o 'conjunto'.
  - La asignación a una variable de tipo 'conjunto de T' es correcta si y solo si la expresión a la derecha es de tipo 'conjunto de T' o 'conjunto'.

- *Ejemplos de expresiones*: Suponiendo lenguaje tipo Pascal con palabras reservadas en inglés.

```
c1 : set of integer ;
c2 : set of character ;

c1 := [ 1,2,3 ] ;
c1 := [ ];
c1 := [ 'a' ] ;     /** error **/

c2 := ['a','b' ] ;
c2 := [] ;
c2 := [1,2,3] ;     /** error **/

(ejemplos de expresiones)

[1,2,3] * [4,5,6]   /** correcta: devuelve {} **/
[1,2,3] * []        /** idem **/
[1,2] * [2,3]       /** correcta: devuelve {2} **/
[1,2] - [2,3]       /** correcta: devuelve {1} **/
[] + []             /** correcta: devuelve {} **/
['a','b'] + [1]     /** error **/
[1] <= [1]          /** correcta: devuelve 'true' **/
[1] >  [1]          /** correcta: devuelve 'false' **/
'a' in ['b', 'c']   /** correcta: devuelve 'false' **/
'a' in [1]          /** error **/
'a' in c            /** correcta si C es de tipo 'set␣of␣char' **/
[a+b, c*3, d]       /** correcta si a,b,c,d son variables de un
                        tipo primitivo **/

a in []             /** correcta si 'a' es una variable de tipo primitivo **/
b in ['a','b']      /** correcta si 'b' es una variable de tipo 'char' **/

[1, 2, 3]*1         /** correcta, de tipo 'conjunto␣de␣enteros'   **/
7*[3, 5, 6]         /** (idem), devuelve {21,35,42}               **/
[]+2.0              /** correcta, de tipo 'conjunto␣de␣flotantes', devuelve {} **/
[1, 2]*7.0          /** error: flotante*(conjunto de enteros)      **/
[1, 2]*'a'          /** error: caracteres no multiplicables       **/
['a', 'b']*2        /** (idem) **/
[7.0, 8.0]/3.0/4.0  /** correcta, de tipo: 'conjunto␣de␣flotantes' **/
                    /** devuelve {7.0/12.0, 8.0/12.0}              **/
```

A.4 Pilas

Este tipo sirve para declarar variables o expresiones que contienen una lista de expresiones mediante una organización tipo LIFO (last in first out) de cualquier tipo base no estructurado.

- *Declaración*: Debe especificarse el tipo base.
- *Ejemplos de declaración*: Dado un lenguaje con palabras reservadas en inglés.

```
/* Lenguaje tipo C */
       stack int   si1, si2 ;
       stack float sf1, sf2 ;
       stack char  sc1, sc2 ;
       stack bool  sb1, sb2 ;
```

```
/* Lenguaje tipo Pascal */
       si1, si2 : stack integer ;
       sf1, sf2 : stack real ;
       sc1, sc2 : stack character ;
       sb1, sb2 : stack boolean ;
```

- *Agregados*: Los agregados del tipo se especifican como una lista (posiblemente vacía) de agregados del tipo base donde la primera será el tope de la pila.
- *Ejemplos de agregados*:

```
{1, 43, 5}                          /** tipo base == entero, tope == 1 **/
{'z', 'A', 'B'}                     /** tipo base == carácter, tope 'z'**/
{2.34+a, 2.222*f(x), 112.00, 12.2} /** tipo base == real, tope 2.34   **/
{true, false, false, true}          /** tipo base == lógico, tope true **/
{}                                  /** Pila vacía, no tiene tipo base **/
```

- *Operadores*: Se definen los siguientes operadores, teniendo en cuenta que son formas de expresión y no de sentencia, es decir, se devuelve un elemento del tipo base de la pila, un valor lógico o una pila tras realizar la operación correspondiente.

Nombre	Tipo	Lexema	Operandos	Ejemplo	Resultado
insertar	binario	$++$	pila y valor	$p++x$	inserta x en el tope de p y devuelve una copia de la propia pila
sacar	unario	&	pila	$\&p$	borra el tope de p y devuelve una copia de la pila p.
tope	unario	#	pila	$\#p$	devuelve el valor del tope de p.
vacía	unario	!	pila	$!p$	devuelve 'true' si la pila está vacía y 'false' en otro caso
producto	binario	$*$	dos pilas	$p_1 * p_2$	producto elemento a elemento
			pila y valor	$p * x$	producto de todos los elementos con x
			valor y pila	$x * p$	producto x con todos los elementos
suma	binario	$+$	dos pilas	$p_1 + p_2$	suma elemento a elemento
			pila y valor	$p + x$	suma de todos los elementos con x
			valor y pila	$x + p$	suma x a todos los elementos
resta	binario	$-$	dos pilas	$p_1 - p_2$	resta elemento a elemento
			pila y valor	$p - x$	resta x a todos los elementos
división	binario	$/$	dos pilas	p_1/p_2	división elemento a elemento
			pila y valor	p/x	divide todos los elementos por x

- *Reglas semánticas*: Cuando se aplica una operación aritmética a dos pilas, ambas deben ser del mismo tipo base y deben tener el mismo número de elementos. Cuando se opera con un dato y una

pila, ambos deben compartir el mismo tipo básico. Por último, cuando se realiza una asignación de la forma:

$$p_1 = p_2;$$

donde p_1 y p_2 son pilas, esa sentencia es válida si y solo si ambas tienen el mismo tipo base. El resultado hace que la pila p_1 tenga los mismos elementos que p_2.

Apéndice B

LEX

B.1 Ejecución de LEX/FLEX

Los programas LEX y FLEX [Lex06, Flex06] son de libre distribución y están disponible tanto en plataformas Windows como Unix/Linux. Su ejecución, según la plataforma es la siguiente:

```
Windows: <Letra unidad LSI>:\lsi\pl\flex -l <resto_opciones> <fichero_descripcion>.lex
Linux:   flex -l <resto_opciones> <fichero_descripcion>.lex
         lex <resto_opciones> <fichero_descripcion>.lex
```

Donde la opción `-l` indica máxima compatibilidad con `lex` original. Entre otras cosas, permite la utilización de ciertas variables predefinidas como por ejemplo `yylineno`, que indica el número de línea del texto fuente.

En Windows se obtiene un fichero C con la implementación en dicho lenguaje del analizador de léxico. Dicho fichero tiene el nombre `lexyy.c` mientras que en Linux el fichero tiene el nombre `lex.yy.c`.

Tal y como se ha mencionado en la práctica anterior, el lenguaje C es sensible a las mayúsculas, cosa que no sucede en Pascal. Para el correcto reconocimiento de patrones (indistintamente con mayúsculas o minúsculas) de elementos del lenguaje Pascal, se debe poner la opción `-i` en `<resto_opciones>`.

B.2 Confección de las especificaciones LEX/FLEX

A la hora de escribir la especificación LEX/FLEX se debe tener en consideración las siguientes recomendaciones:

1. Definir los patrones en orden de menor a mayor generalidad. En primer lugar se especifican las palabras reservadas y por último las constantes y el identificador.
2. Colocar cada lexema (palabra) de un mismo token en líneas distintas, aunque devuelva el mismo código de token.
3. El último patrón debe ser el ".", cuya acción es informar de la presencia de errores léxicos, y por tanto, no debe devolver nada.

Apéndice C

YACC

C.1 Ejecución de YACC

YACC/BYACC/BISON son alternativas de uso:

- YACC: Implementación estándar para plataformas Unix/Linux [Yacc06].
- BYACC: Implementación de libre distribución realizada en la Universidad de Berkeley compatible con YACC disponible para plataformas Dos/Windows e incluso para Linux.
- BISON: Implementación de libre distribución realizada por GNU compatible con los dos anteriores y disponible para plataformas Dos/Windows y Unix/Linux [Biso06].

Al invocar a YACC debemos indicar el fichero fuente, normalmente acabado en `.y`, que contiene la definición de la gramática. Como resultado (si no hay errores) se producirá un programa (escrito en lenguaje C ANSI) almacenado en el fichero `y_tab.c` (plataformas Dos/Windows) o `y.tab.c` (plataformas Unix/-Linux) en nuestro directorio de trabajo. Entre otros, podemos incluir los siguientes modificadores al ejecutar YACC:

`-t` Incluir código de depuración en el programa generado. Esto permite que el analizador obtenido, al ser ejecutado, escriba en pantalla las producciones que va aplicando cuando procesa una entrada. Para hacerlo, debemos fijar la variable de entorno `YYDEBUG` a `1` antes de ejecutar el analizador. Poniéndola a 0 (por defecto) no se obtendrá salida de depuración.

`-v` Indica a YACC que genere, además de `y_tab.c`/`y.tab.c`, el fichero `y.out` (plataformas Dos/Windows) o `y.output` (plataformas Unix/Linux), que contiene una descripción textual de los estados del autómata finito determinista y la tabla de análisis corresondientes a la gramática de entrada.

`-d` Indica a YACC que genere el archivo `y_tab.h` (plataformas Dos/Windows) o `y.tab.h` (plataformas Unix/Linux), que contiene la lista de tokens junto con su código. Esto es especialmente útil para comprobar errores de tratamiento de los tokens por considerar un código incorrecto.

C.2 Uso conjunto de YACC y LEX

Para la realización de esta práctica es necesario usar el analizador sintáctico y el léxico en combinación. Una opción para hacer esto es estructurar los documentos fuentes YACC y LEX como en el ejemplo dado al

final de esta sección, en los cuales se incluyen los ficheros `prueba.l`, `prueba.y` y `main.c`. Respecto a estos ficheros, hay que tener en cuenta los siguientes aspectos:

- La definición de los tokens ahora se encuentra en el fichero fuente de YACC y ha sido eliminada del fuente LEX. Esto se hace así para poder incluir los nombres simbólicos de los tokens en las producciones de la gramática.
- El fichero `lex.yy.c / lexyy.c` se incluye al final del documento de especificación YACC. De esta forma, podemos usar los mismos nombres simbólicos de los tokens en ambos fuentes.
- Para obtener el procesador será necesario compilar `y.tab.c / y_tab.c` y `main.c`. Posteriormente, se enlazarán los objetos obtenidos.
- La función `abrir_entrada` incluida en el módulo `main.c` permite que se lea bien de la entrada estándar (si no se da ningún nombre) o bien de un fichero si se da como argumento.
- La función `yyerror` será invocada siempre que ocurra un error sintáctico en la entrada. Además, la hemos usado también para los errores léxicos en `prueba.l`.

A continuación se muestra un ejemplo de fuente en BISON (fichero `prueba.y`).

```
%{
/*******************************************************
 **
 **  Fichero: PRUEBA.Y
 **  Función: Pruebas de YACC para practicas de PL
 **
 *******************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/**  La siguiente declaracion permite que 'yyerror' se pueda invocar desde el
***  fuente de lex (prueba.l)
**/

void yyerror( char * msg ) ;

/**  La siguiente variable se usará para conocer el numero de la línea
***  que se esta leyendo en cada momento. También es posible usar la variable
***  'yylineno' que también nos muestra la línea actual. Para ello es necesario
***  invocar a flex con la opción '-l' (compatibilidad con lex).
**/

int linea_actual = 1 ;
%}

/**  Para uso de mensajes de error sintáctico con BISON.
***  La siguiente declaración provoca que 'bison', ante un error sintáctico,
***  visualice mensajes de error con indicación de los tokens que se esperaban
***  en el lugar en el que produjo el error (SÓLO FUNCIONA CON BISON>=2.1).
***  Para Bison<2.1 es mediante
***
***  #define YYERROR_VERBOSE
***
***  En caso de usar mensajes de error mediante 'mes' y 'mes2' (ver apéndice D)
***  nada de lo anterior debe tenerse en cuenta.
**/

%error-verbose

/**  A continuación declaramos los nombres simbólicos de los tokens.
***  byacc se encarga de asociar a cada uno un código
**/

%token  NOMBRE VERBO
%token  NOMBRE_PROPIO
```

```
%token  ARTICULO ARTICULO_A
%token  DESCONOCIDO

%%

/**  Sección de producciones que definen la gramática.
**/

frase              : sintagma_nominal sintagma_predicado ;

sintagma_nominal   : ARTICULO NOMBRE
                   | NOMBRE_PROPIO ;

sintagma_predicado : VERBO modificador objeto_directo ;

objeto_directo     : sintagma_nominal ;

modificador            : ARTICULO_A
                   | /** esto representa la cadena vacía **/ ;

%%

/** aqui incluimos el fichero generado por el 'lex'
*** que implementa la función 'yylex'
**/

#ifdef DOSWINDOWS          /* Variable de entorno que indica la plataforma */
#include "lexyy.c"
#else
#include "lex.yy.c"
#endif

/**  se debe implementar la función yyerror. En este caso
***  simplemente escribimos el mensaje de error en pantalla
**/

void yyerror( char *msg )
{
  fprintf(stderr,"[Linea %d]: %s\n", linea_actual, msg) ;
}
```

A continuación se muestra un ejemplo de fuente FLEX (fichero `prueba.l`).

```
%{
/*******************************************************
 **
 ** Fichero: PRUEBA.L
 ** Función: Pruebas de FLEX para practicas de PL.
 **
 *******************************************************/

/**  Buffer usado para almacenamiento temporal de cadenas de texto
**/

#define MAXLONBUFFER 200
char buffer[MAXLONBUFFER] ;

%}

/**  La siguiente declaración indica que no se va a escribir la función 'yywrap'
***  evita que se tenga que enlazar con -lfl, o que se tenga que escribir 'yywrap'
**/

%option noyywrap

%%

el|la            {  return(ARTICULO)    ;   }
a                {  return(ARTICULO_A)  ;   }
habla|ve|mira    {  return(VERBO) ;         }
Juan|Pepe|Maria  {  return(NOMBRE_PROPIO) ; }
perro|gato|luna  {  return(NOMBRE) ; }

\n        {  /** lleva la cuenta de las líneas **/
```

```
              linea_actual++ ;
                      }

[ \t]+    {  /** accion vacia: se ignoran los espacios  y tabuladores **/
          }

.         {  /** token desconocido, se produce un mensaje de error **/

              snprintf(buffer, MAXLONBUFFER, "el(los) caracter(es) '%s' no
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣forma(n) ningun token conocido",yytext);
              yyerror(buffer);
          }
```

A continuación se muestra un ejemplo del módulo MAIN (fichero `main.c`).

```
/*********************************************************
 **
 ** Fichero: MAIN.C
 ** Función: Pruebas de FLEX para practicas Proc.Leng
 **
 *********************************************************/
#include <stdio.h>
#include <stdlib.h>

extern FILE *yyin ;
int yyparse(void) ;

FILE *abrir_entrada( int argc, char *argv[] )
{
  FILE *f= NULL ;

  if ( argc > 1 )
  {
     f= fopen(argv[1],"r");
     if (f==NULL)
     {
        fprintf(stderr,"fichero '%s' no encontrado\n",argv[1]);
        exit(1);
     }
     else
        printf("leyendo fichero '%s'.\n",argv[1]);
  }
  else
     printf("leyendo entrada estándar.\n");
  return f ;
}

/*********************************************************/

int main( int argc, char *argv[] )
{
  yyin= abrir_entrada(argc,argv) ;
  return yyparse() ;
}
```

Finalmente, un ejemplo de archivo `Makefile` para poder realizar todo el proceso de invocación a `flex`, `bison` y la compilación con `gcc` podría ser:

```
.SUFFIXES:

prueba: main.o y.tab.o
        gcc -o prueba main.o y.tab.o

y.tab.o: y.tab.c
        gcc -c y.tab.c

main.o: main.c
        gcc -c main.c

y.tab.c: prueba.y lex.yy.c
        bison -o y.tab.c prueba.y

lex.yy.c: prueba.l
```

```
	flex prueba.l

limpia:
	rm -f prueba main.o y.tab.o y.tab.c lex.yy.c

todo:
	make limpia
	make prueba
```

Apéndice D

Consideraciones sobre Análisis Sintáctico

D.1 Precedencia y Asociatividad de Operadores Comunes

Para evitar conflictos en la tabla de análisis es necesario la determinación de la precedencia y asociatividad de los operadores. En la siguiente tabla se muestran los operadores comunes junto a su precedencia y asociatividad ordenados de menor a mayor. Aquellos que se sitúan en la misma línea indican igualdad de precedencia[1]

Operador	Propósito	Asociatividad
$\|\|$	OR lógico	izquierda
&&	AND lógico	izquierda
$\|$	OR bits	izquierda
$\wedge$	Exclusive OR	izquierda
&	AND bits	izquierda
$== !=$	Operadores de igualdad	izquierda
$<><=>=$	Operadores de relación	izquierda
$+ -$	Operadores aditivos	izquierda
$* / \%$	Operadores multiplicativos	izquierda
$* \& + - !$	Operadores unarios	derecha
$--$	Decremento (prefijo)	derecha
$++$	Incremento (prefijo)	derecha
$--$	Decremento (posfijo)	izquierda
$++$	Incremento (posfijo)	izquierda
[]	Índices	izquierda

A partir de esta tabla se deben añadir a consideración del grupo de alumnos las precedencias y asociatividades de los operadores que surjan de implementar el tipo de dato estructurado.

[1] Tabla obtenida a partir de la precedencia y asociatividad de los operadores en C++.

D.2 Estilo de las Producciones

Al implementar YACC un analizador ascendente, suele ser preferible que las producciones sean recursivas a izquierda. En el caso típico de listas de elementos, esto permite que las reducciones se vayan realizando al leer cada elemento. Si se usa recursividad a derechas, las reducciones solo pueden realizarse una vez leídos todos los elementos. A modo de ejemplo, siempre es más óptimo producciones de la siguiente forma:

```
Lista : Lista Elemento
      | ;
```

frente a producciones menos eficientes como

```
Lista : Elemento Lista
      | ;
```

En este ultimo caso, el numero de entradas de la pila ocupadas es igual al número de elementos de la lista, con lo cual se usa mucha más memoria que en la primera opción. Al usar recursividad a la izquierda, la reducción por la producción `Lista :` (producción a la cadena vacía) se realiza al inicio del análisis de la lista. Si usásemos recursividad a la derecha, esta reducción solo puede hacerse al final de la lista.

D.3 Tratamiento de Errores

En esta sección se muestran los dos aspectos a considerar en cuanto al tratamiento de errores. Por un lado, cuando se detecta un error debe ser informado mediante un mensaje que pueda ser lo suficientemente claro para que el programador pueda descubrir el lugar donde se produce (informar del número de línea de código donde se produce) y poder arreglarlo posteriormente. Por otro lado, es ideal que el traductor detecte el mayor número posible de errores e informar de ellos con objeto de corregirlos posteriormente en un nuevo proceso de edición del texto fuente.

En la sección D.3.1 se muestra cómo es posible informar del error producido usando `bison` y en la sección D.3.2 se muestra la estrategia a seguir para recuperarnos ante un error y continuar el análisis sintáctico para detectar e informar de posibles nuevos errores, si los hubiera.

D.3.1 Mensajes de Error

El análisis ascendente LR detecta un error cuando el estado en el tope de la pila y el símbolo de entrada le corresponde una celda vacía en la tabla de análisis LR. En base a la información de la tabla de análisis, para cada estado se dispone de los símbolos de entrada esperados (aquellos símbolos con acciones en la tabla) y el resto de símbolos de entrada considerados como entradas erróneas. Esto nos permite generar un mensaje de error para cada estado de la tabla de análisis en los términos siguientes:

Supongamos la siguiente tabla de análisis:

	PARIZQ	PARDCH	IDENT	ASIGN	IF	$
...						
6		s3		r4	s5	
...						

Donde PARIZQ es '(', PARDCH es ')', IDENT es cualquier identificador, ASIGN es el operador de asignación '=' e IF es el token de la sentencia condicional.

Se produciría error si en el tope de la pila aparece el estado 6 y en la entrada aparece el símbolo (, IDENT o $. En ese caso, si aparece en la entrada el token **IDENT** y el 6 es el estado del tope de la pila, se debe emitir el siguiente mensaje de error:

```
Error en línea 1: syntax error, unexpected 'IDENT', expecting PARDCH or ASIGN or IF
```

Mensajes de Error con BISON

Estos mensajes de error pueden obtenerse de la tabla de análisis automáticamente incluyendo la declaración **`%error-verbose`** en la especificación YACC, dentro de la zona de declaraciones y usando **bison 2.1** en lugar de yacc/byacc:

```
...
%error-verbose
...
Zona de Declaraciones
%%
Reglas gramaticales y Acciones YACC
%%
Resto de subprogramas
```

Para aquellos que usen una versión anterior (usando `bision -version` se puede conocer la versión que se ejecuta) deben declararlo de la siguiente forma:

```
...
%{
#define YYERROR_VERBOSE
%}
...
Zona de Declaraciones
%%
Reglas gramaticales y Acciones YACC
%%
Resto de subprogramas
```

Teniendo en cuenta los ejemplos mostrados en el apéndice anterior, se muestran a continuación una serie de frases de entrada que contienen errores léxicos y sintácticos donde se aprecia el mensaje de error que se comunica:

```
./prueba
leyendo entrada estándar.
%Juan habla a Maria
Error en linea 1 leyendo '%'

Juan el habla a Maria
Error en linea 1:  syntax error, unexpected ARTICULO, expecting VERBO

Juan Maria habla
Error en linea 2:  syntax error, unexpected NOMBRE_PROPIO, expecting $end

Ju%an habla a Maria
Error en linea 1 leyendo 'J'
Error en linea 1 leyendo 'u'
Error en linea 1 leyendo '%'
Error en linea 1:  syntax error, unexpected ARTICULO_A, expecting NOMBRE_PROPIO or ARTICULO
```

Mensajes de Error con YACC/BYACC

En el caso de YACC/BYACC no existe una forma automática como lo mostrado en la sección anterior. En este caso se proporciona una especificación léxica llamada `mes.l` y el ejecutable para entornos Windows `mes.exe`. En entornos Unix/Linux es necesario obtener el ejecutable de la siguiente forma:

```
% flex -l mes.l
% gcc -omes lex.yy.c
```

Posteriormente, el archivo de mensajes de error se genera a partir de la tabla de análisis obtenida por YACC. El proceso sería el siguiente:

```
mes < y.output > msj.err    (para entornos Unix/Linux)
mes < y.out > msj.err       (para entornos Dos/Windows)
```

Donde `y.out` o `y.output` es la salida del YACC con la opción `-v` y el archivo `msj.err` es el fichero que alberga un array cuya dimensión es el número de estados de la tabla de análisis proporcionada por YACC y su contenido son mensajes de error de la forma siguiente:

```
char *msjerror[] = { "Error(0): esperaba .....",
                     "Error(1): esperaba .....",
                         ....
                   };
```

El **array de errores debe generarse cada vez que se altera la gramática**, dado que cambiaría el número de estados y, por tanto, la tabla de análisis LALR que genera YACC. Dicho array puede incluirse en la especificación YACC de la siguiente forma:

```
Zona de Declaraciones de tokens
%%
Reglas gramaticales y Acciones YACC
%%
...
#include "msj.err"
...
int main ()
{
   ...
}
```

Por defecto, cuando se alcanza un error sintáctico, el programa salta a la etiqueta del programa `yynewerror` en donde se ejecuta la función `yyerror ("Syntax error");`. Dicha función emite ese mensaje de error ante cualquier error sintáctico. Si queremos que nos emita el mensaje de error concreto tal y como hemos confeccionado el array de mensajes de error, debemos sustituir esta llamada por una de la forma `yyerror (msjerror[yystate]);`, donde `yystate` es el estado donde se ha quedado el analizador cuando encontró un error sintáctico.

Para poder usar el array de mensaje de error en el código del analizador sintáctico generado por YACC, se ha creado la utilidad `mes2.l` cuya función es sustituir la llamada a error sintáctico con el mensaje por defecto por el indicado para mostrar el texto del mensaje de error según el estado donde se encontró dicho error, es decir, usando el array de mensajes de error.

Antes que nada, es necesario obtener el ejecutable de dicho archivo y luego pasárselo al código del analizador sintáctico para que altere la llamada a la función que emite el mensaje de error. El proceso completo sería el siguiente:

```
% flex -l mes2.l
% gcc -omes2 lex.yy.c
...
% yacc -dtv sintactico.y     (genera el analizador por defecto en y.tab.c)
% mes2 < y.tab.c > y.tab2.c  (genera y.tab2.c con mensaje de error específicos)
```

Tanto `mes.l` como `mes2.l` están disponibles en la web de la asignatura, dentro del apartado de Software.

D.3.2 Recuperación ante un Error Sintáctico

YACC admite un tipo especial de producciones que permiten introducir acciones de tratamiento de error y recuperarse ante el mismo en *modo panic*. Para ello existe un token predefinido cuyo nombre es `error`. Este token es incluido en la entrada cada vez que el analizador se encuentra en una situación de error. En concreto, supongamos que `nt` es un símbolo no terminal y que `t` es un símbolo terminal cualquiera. En estas condiciones, las producciones de la forma:

```
nt : error t { <accion> }
```

indican que ante una situación de error, primero invoca la función `yyerror` que nos informará del error sintáctico y después aplica el proceso de recuperación ante el error. Para ello, sacará símbolos de la pila hasta que alcance un estado que disponga de la acción desplazar con `error` y a continuación descartará símbolos de la entrada hasta encontrar el terminal `t` que será desplazado, apareciendo la parte derecha de la producción anterior en el tope de la pila, y por consiguiente, se produce una reducción, reemplazando el tope de la pila por el símbolo no terminal `nt`. Después de recuperarse ante el error se termina el proceso de análisis. Si queremos que el proceso de análisis siga, debemos incluir la sentencia `yyerrok` (como está, sin paréntesis) en la acción (`nt:error t {yyerrok;}`).

¿Es obligatorio colocar un símbolo terminal justo después del token predefinido `error`? La respuesta es que no. En ese caso, sigue saltando símbolos de la pila hasta alcanzar un estado con transición de desplazar con el token `error`, sólo que en esta ocasión no saltará símbolos de la entrada hasta alcanzar uno en concreto (como en el caso anterior) sino que saltará símbolos hasta que aparezca en la entrada un seguidor del símbolo no terminal.

De este modo, otra forma de recuperarse ante el error sería sincronizar con los símbolos seguidores de `nt`. Para ello bastaría con definir la producción de la siguiente forma:

```
nt : error ;
```

Como se aprecia, ni aparece símbolo de sincronización ni acción. En este caso cuando encuentra un error el analizador se recupera como si estuviese sincronizado por los seguidores del símbolo no terminal `nt`, continuando el análisis sin necesidad de la acción `yyerrok`. **Esta última forma es la recomendada para realizar la práctica y sólo en caso de problemas se usaría la anterior**[2].

Debemos tener en cuenta que `error` es considerado por YACC como un símbolo terminal. Si incluimos producciones de error que tengan (directa o indirectamente) la siguiente estructura:

```
A : B
  | C ;
B : error ;
C : error ;
```

[2] Hasta ahora, dadas las recuperaciones de error que deben considerarse en la práctica 3 de análisis sintáctico, en ningún tipo de lenguaje ha sido necesario recurrir a la primera de las opciones para la recuperación ante el error.

entonces obtendremos conflictos reduce/reduce, ya que ante una situación de error, el analizador podría reducir tanto por la segunda regla como por la tercera. En realidad, en esta situación, el analizador podría no tiene información suficiente para saber si realmente el programador quería escribir algo concordante con `B` o con `C`. Esto debe ser evitado asociando siempre producciones de error a símbolos no terminales que no sean alternativas de otro no terminal a más alto nivel.

Un uso correcto de producciones de `error` sería como sigue (asumiendo que `x` e `y` son dos símbolos terminales distintos):

```
A : x B
  | y C ;
B : error ;
C : error ;
```

En este caso no habría conflictos, ya que los símbolos iniciales distintos permiten discriminar si se ha entrado en una zona concordante con `B` o con `C`, y realizar las recuperaciones necesarias en cada caso.

Es competencia de cada grupo de prácticas elegir el lugar más adecuado en donde la producción alternativa error debe incluirse para que satisfaga la prueba de la práctica con éxito.

No obstante, sugerimos que se estudie las producciones que definen frases (sentencias, declaraciones de variables, definiciones de funciones/procedimientos etc.) como las candidatas a disponer de la alternativa de error, ya que son las sentencias simples lo máximo que estamos dispuestos a perder en caso de error (recuerde que la producción `nt:error` conlleva que en caso de error se perdería todo lo que pueda derivarse del símbolo `nt`). De igual modo, las frases compuestas debería asegurarse que todas sus componentes serán tratadas de la forma deseada ante el error.

Apéndice E

Ejemplo de Código para Prueba de Sintáctico/Semántico

Ejemplo: Dado un lenguaje de componentes BBDAC, es decir, basado en C con palabras reservadas en inglés, pila como dato estructurado, funciones y la estructura de control adicional repeat-until, este podría ser un ejemplo de código para la prueba de la práctica:

```
main()
{
  var
     int   ve ;
     float vf ;
     char  vc ;
     bool  vl ;
     stack int   pe, pe2 ;
     stack float pf, pf2 ;
     stack char  pc, pc2 ;
     stack bool  pl ;
  endvar

  int funcionA (int a1, float a2, char a3)
  {
     var
        int x1, x2 ;
     endvar

        char funcionB (char b1, bool b2)
        {
           var
              float xf, x2 ;
           endvar

              float funcionC (bool c1, int c2)
              {
                 var
                    float x1 ;
                 endvar

                 x1= 1.3 ;
                 if (c2>10)
                   c2= c2-1 ;
                 else
                   x1= 3.1 ;
                 return x1 ;
              }

           xf= functionC (true, 10);
           x2= xf*(funcionC(false,1)-funcionC(true,23))/10.0;

           while (x2*funcionC(false,1)-xf<10.0)
              x2= x2*xf ;
        }

        float funcionD (float d1)
```

```
       {
          var
             char dato ;
             int valor ;
          endvar

             char funcionE (char e1, char e2)
             {
                 read "introduzca␣dos␣caracteres:␣", e1, e2 ;
                 if (e1=='a')
                    return e1 ;
                 else if (e1=='b')
                    return e2 ;
                 else
                    return '␣';
             }

          read "introduzca␣un␣valor␣entero:␣", valor ;
          if (d1>0.0)
          {
             var
                int dato ;
             endvar

             dato= 2 ;
             dato= valor*20/dato ;
          }
          else {
             valor= valor * 100 ;
             d1= d1/1000.0 ;
          }
          return d1 ;
       }

    pe= pe<-10 ;     /* Inserta 10 en la pila 'pe' */
    pf= pf<-10.0 ;   /* Inserta 10.0 en la pila 'pf' */
    pc= pc<-'#' ;    /* Inserta # en la pila 'pc' */

    if (?(pe<-20) == 20)  /* Inserta 20 y si el tope de la pila 'pe' es 20 */
       ve= pe-> ;         /* Extrae el tope de la pila y lo guarda en 've' */
    else
       pe= pe * pe2 ;     /* Multiplica pe y pe2 y lo almacena en pe */
       pe= pe2 - pe <- 10 * (20/2000) ;
}
```

Apéndice F

Análisis Semántico

F.1 Estructuras de la Tabla de Símbolos

Por tabla de símbolos (TS) entenderemos una estructura de datos que contendrá, en cada momento, la información necesaria acerca de todos los identificadores (o símbolos) cuyo ámbito incluya el punto actual de compilación (esto es, todos los identificadores que se pueden usar). Por información necesaria entendemos todos los datos requeridos para hacer las comprobaciones semánticas. Esta información se agrupará en forma de registro, al que denominaremos entrada, y que tendrá una estructura fija independientemente del tipo de identificador que describe.

La anterior definición hace necesario incluir en la TS una entrada cada vez que un nuevo identificador es declarado, así como eliminar de la TS una entrada cuando finalice el ámbito del identificador que describe. Lo primero ocurre siempre en la sección de declaraciones de un bloque, y lo segundo al final de los bloques. Todo esto nos lleva a estructurar la TS como una pila de entradas. En ella metemos y sacamos entradas a medida que comienzan o terminan los ámbitos de los identificadores. Al encontrarnos con la declaración de una variable, construimos una nueva entrada y hacemos un push de ella en la TS Cuando un bloque finaliza, debemos hacer pop de todas las entradas de identificadores declarados en dicho bloque.

Para realizar esto último, podemos definir un tipo de entrada especial que marca en la pila el comienzo de un bloque, con lo cual al desactivar el bloque eliminaremos todas las entradas hasta la primera marca que encontremos. Al entrar en un bloque, lo primero que haremos será un push de esta marca en la pila.

Algunas entradas llevarán información sobre un identificador que referencia a un procedimiento o función, y por tanto será necesario incluir información sobre sus parámetros. Para realizar esto, se incluirán los nombres y tipos de los parámetros formales de un subprograma en las sucesivas entradas de la tabla de símbolos posteriores a la correspondiente a una función. Estas entradas especiales estarán marcadas para indicar que no corresponden a identificadores activos. Se usarán al comprobar los parámetros actuales de las llamadas a subprogramas con respecto a los parámetros formales indicados en su declaración.

Algo parecido ocurre con la lista de rangos de una matriz, en el caso de matrices multidimensionales. En la entrada correspondiente a la matriz incluiremos el rango de la primera dimensión. En el caso de que el número de dimensiones sea mayor que 1, en las sucesivas entradas se indicarán los rangos de las sucesivas dimensiones. Estas otras entradas especiales estarán debidamente marcadas.

Según el tipo de lenguaje, existen dos opciones de estructura de la sección de declaración de variables, que son:

- En algunos casos, la lista de identificadores precede a la descripción del tipo de la variable. Al incluir

una entrada en la tabla de símbolos, no sabremos aún su tipo. Por tanto será necesario incluirla con una marca especial, que indica que el tipo aún no ha sido asignado. Al procesar la declaración del tipo, se deben actualizar las entradas de la tabla de símbolos cuyo tipo no hubiese sido asignado aún.

- En otros casos, la lista de identificadores va después de la descripción del tipo. Aquí debemos almacenar el tipo en una variable intermedia (que llamaremos `TipoTmp`). Después, según van apareciendo los identificadores, se dan de alta en la tabla de símbolos, tomando el tipo de dicha variable intermedia.

Todas las consideraciones anteriores nos llevan a definir las entradas de la TS como una estructura con los siguientes componentes:

TipoEntrada: Indica el tipo de entrada. Este valor será de un tipo enumerado, con los siguientes valores posibles:

`marca`	Indica que la entrada es una marca de principio de bloque.
`procedimiento`	La entrada describe un procedimiento.
`funcion`	La entrada describe una función.
`variable`	La entrada describe una variable local.
`parametro-formal`	La entrada describe un parámetro formal de un procedimiento o función situado en una entrada anterior de la tabla.

Nombre: Un puntero a una cadena de caracteres que contiene los caracteres que forman el identificador. En el caso de que *TipoEntrada* sea `marca`, no se usará.

TipoDato: En el caso de que *TipoEntrada* sea `funcion`, `variable` o `parametro-formal`, indica el tipo del adato al que se hace referencia, en otro caso no se usa. Será un tipo enumerado cuyos valores posibles son uno de entre los siguientes:

`booleano`	Tipo básico lógico o booleano.
`entero`	Tipo básico entero.
`real`	Tipo básico real o coma flotante.
`caracter`	Tipo básico carácter.
`array|pila|conjunto|lista`	Tipo estructurado asignado.
`desconocido`	La entrada describe el rango en una dimensión de un array situado en una entrada anterior de la tabla.
`no-asignado`	Para lenguajes en los que las declaraciones de variables, el tipo vaya detrás de la lista de identificadores (caso de Pascal), indica que aún no se ha alcanzado la descripción del tipo y, por tanto, aún no se conoce el tipo para esta entrada.

Parametros: Es un valor entero positivo que indica el número de parámetros formales, solo si el tipo es `procedimiento` o `funcion`, en otro caso no se usa.

TamañoDimension1 y TamañoDimension2: Son dos valores enteros que indican el tamaño de las dimensiones. Sólo tiene sentido si *TipoDato* es `array`.

Una vez definidas las entradas, podemos definir la TS como un array unidimensional de entradas como la descrita (lo llamaremos TS). Este array tendrá un tamaño máximo fijo predefinido, que se hará lo suficientemente grande como para que se pueda procesar cualquier programa de tamaño de prueba (no superior

a 1000 líneas). Además del array, existirá una variable entera `TOPE` inicializada a 0 y que indica en cada momento la última entrada del array usada. `TOPE` debe declararse como un entero largo (`long int`). *En cualquier caso, si se llena la TS debería dar un error*.

Para introducir una entrada en TS, bastará incrementar el valor de `TOPE` en uno y después asignar valores a los elementos de la entrada `TS[TOPE]`. Para extraer el último elemento de TS, solo es necesario disminuir tope en uno. En el caso de que `TOPE` alcance el valor máximo prefijado, se producirá un error y se abortara la compilación. Para ello se debe definir una función mediante la cual se incremente `TOPE`, realizando la comprobación descrita.

F.2 Acciones Semánticas con uso de la Tabla de Símbolos

Una vez definidas las estructuras de datos necesarias, veremos cuales son las acciones semánticas asociadas a las reglas sintácticas.

1. En el análisis de léxico, al reconocer un identificador, se debe de guardar en `yylval` un puntero a la cadena de caracteres que lo forman, para que el analizador sintáctico tenga acceso a esa información. Para realizar ésto, habría que guardar en el campo cadena de `yylval` una copia (obtenida con `strdup`) del contenido de `yytext` (es importante no usar referencias directas a `yytext`, puesto que su contenido se reescribe cada vez que se reconoce un token en la entrada). En el fuente YACC, este valor es leído como el atributo sintetizado por el símbolo terminal "identificador"

2. Al reconocer un identificador en una declaración de variables es necesario introducir una nueva entrada en TS. Si hubiese otro identificador con el mismo nombre y posterior a la última marca de bloque, se produce un error de identificador duplicado.

3. Al entrar en un bloque, se debe incluir en la tabla de símbolos una marca de inicio de bloque. En el caso de que el bloque corresponda al cuerpo de un subprograma, los parámetros formales (que se encuentran por encima de la marca en la tabla de símbolos) deberán ser introducidos de nuevo en TS como si fuesen variables locales, ya que no se pueden definir variables con el mismo nombre que los argumentos.

4. Al reconocer el tipo en una declaración de variables, será necesario asignar tipo a las variables de TS que no lo tuviesen asignado (para lenguajes con el tipo después de los nombres de las variables) o bien asignar la variable temporal de transito `TipoTmp` con este tipo. Para lo anterior será necesario que el no terminal asociado con la descripción de tipo tenga como atributo sintetizado precisamente el registro descriptor de tipo `dtipo` dentro de la estructura `atributos`.

5. Al alcanzar el final de un bloque, se deben eliminar de TS todas las entradas hasta la última marca de inicio bloque, incluida ésta.

6. Al reconocer un identificador que contenga el nombre de un procedimiento, en su declaración, se insertará en la tabla de símbolos una nueva entrada. El número de parámetros se pondrá inicialmente a cero.

7. Al reconocer un identificador en la lista de parámetros formales de la cabecera de un procedimiento o función, se insertará una entrada en la TS, con el nombre y tipo del identificador. Además, será necesario incrementar el número de parámetros de dicho subprograma, que se encontrará en una entrada anterior de la TS.

F.3 Estructuras de los Atributos Sintetizados

Al usar YACC, disponemos de una pila (gestionada internamente) de atributos sintetizados por los símbolos de la gramática. El usuario de YACC tiene la posibilidad de redefinir este tipo de dato para adaptarlo a sus necesidades. En nuestro caso, usaremos los atributos sintetizados fundamentalmente con dos fines:

- Para informar al analizador sintáctico de los atributos de los tokens que encuentra el analizador léxico (a través de la variable `yylval`). En este caso, los atributos sintetizados pueden ser de los siguientes tipos:
  - Puntero a una cadena de caracteres, para identificadores y literales.
  - Valores enteros, para constantes enteras y booleanas.
- Para sintetizar el tipo en las expresiones, términos y variables de asignaciones (a través de `$$`). Aquí necesitaremos las suficiente información para describir completamente el tipo de una expresión.

Con estas consideraciones, podremos definir los atributos sintetizados como una estructura tipo `struct` de C, que contiene cada una de las alternativas posibles detalladas arriba. En el fuente YACC incluiremos la directiva de redefinición del tipo de la pila. De esta forma, las variables propias de las acciones semánticas de YACC `yylval` y las pseudovariables `$$`, `$0`, `$1`, etc. serán del tipo `atributos`. Respecto al campo `dtipo`, incluirá un valor de un tipo enumerado para indicar el tipo básico de la expresión (entero, real, booleano, carácter, conjunto, pila, array, lista). Podemos usar el mismo tipo enumerado que usamos en el campo *TipoDato* en las entradas de la TS.

En la mayoría de los casos, esta información es suficiente. En el caso de array, necesitamos además incluir información sobre el número de dimensiones y la lista de rangos posibles del índice en cada dimensión. Para hacer ésto podemos usar listas enlazadas en memoria dinámica y almacenar dentro de los registros `dtipo` un puntero a la cabecera de dicha lista.

F.4 Acciones para las Comprobaciones Semánticas

1. Todos los identificadores detectados deben aparecer en la TS.
2. En las sentencias de control de flujo, como `if`, `while`, etc.. se debe comprobar que la expresión que forma la condición es de tipo booleano.
3. En las sentencias de asignación, el tipo de la variable en la parte izquierda debe coincidir con el tipo de la expresión en la parte derecha. Esto es también así para las asignaciones al contador en las sentencias tipo `for`.
4. En las expresiones compuestas de subexpresiones o términos relacionados con operadores, se debe comprobar que los tipos son concordantes, y se debe sintetizar el tipo ascendentemente. Esto se realiza dando valor al campo `dtipo` de la pseudovariable `$$`.

F.5 Aspectos Básicos de Implementación

A continuación se muestran indicaciones y sugerencias para el comienzo de la práctica. Obviamente, debe entenderse esta parte del apéndice como una muestra de cómo se podría implementar la práctica como

sugerencia. Otros aspectos deben entenderse como "detalles de implementación".

En las siguientes secciones se detallarán las estructuras de datos definidas con anterioridad y cómo aplicar las comprobaciones semánticas en las fases donde se deben realizar (léxico, sintáctico y semántico). En la sección F.5.1 se muestran las estructuras de datos de manejo de una tabla de símbolos, mientras que en las secciones F.5.2 F.5.3 y F.5.4 se muestran los cambios a realizar en dichas fases para la correcta implementación del analizador semántico.

F.5.1 Estructuras de Datos

tipoEntrada

```
typedef enum {
    marca,                    /* marca comienzo bloque */
    <procedimiento,funcion>,  /* si es subprograma */
    variable,                 /* si es variable */
    parametro_formal,         /* si es parametro formal */
} tipoEntrada ;
```

dtipo

```
typedef enum {
    entero,
    real,
    caracter,
    booleano,
    <array,pila,conjunto,lista>
    desconocido,
    no_asignado
} dtipo ;
```

entradaTS

```
typedef struct {
    tipoEntrada  entrada ;
    char         *nombre ;
    dtipo        tipoDato ;
    unsigned int parametros ;
    unsigned int dimensiones ;
    int          TamDimen1 ;   /* Tamaño Dimensión 1 */
    int          TamDimen2 ;   /* Tamaño Dimensión 2 */
} entradaTS ;
```

Resto de Definiciones

```
#define MAX_TS 500

unsigned int TOPE=0 ;      /* Tope de la pila */
unsigned int Subprog ;     /* Indicador de comienzo de bloque de un subprog */

entradaTS TS[MAX_TS] ;     /* Pila de la tabla de símbolos */

typedef struct {
     int    atrib ;        /* Atributo del símbolo (si tiene) */
     char  *lexema ;       /* Nombre del lexema */
     dtipo  tipo ;         /* Tipo del símbolo */
} atributos ;

#define YYSTYPE atributos  /* A partir de ahora, cada símbolo tiene */
                           /* una estructura de tipo atributos */

/* Lista de funciones y procedimientos para manejo de la TS */
    ...
/* Fin de funciones y procedimientos para manejo de la TS */
```

F.5.2 Análisis Léxico

1. Se deben modificar las entradas a ciertos patrones para poder recuperar información de cara a la tabla de símbolos y para la posterior generación de código. Antes de devolver el código del token reconocido, en algunos casos será preciso almacenar el lexema o su atributo de cara a confeccionar la entrada en la tabla de símbolos. Recordar que la variable `yylval` está definida por defecto en LEX y YACC y es del tipo YYSTYPE.

Cambios a nivel de Especificación LEX

```
...
{identificador}         { yylval.lexema = strdup (yytext) ;
                          return IDENT ;
                        }
...
{constante}             { yylval.atrib = <codigo-atrib-constante> ;
                          return CONST ;
                        }

...
{operador_relacional}   { yylval.atrib = <codigo-atrib-operador> ;
                          return OP_REL ;
                        }
...
{tipo_variable}         { yylval.atrib = <codigo-atrib-variable> ;
                          return TIPOVAR ;
                        }
...
```

F.5.3 Análisis Sintáctico

1. Al reconocer un identificador en una declaración de variables, es necesario introducir una nueva entrada en la tabla de símbolos. Si hubiese otro identificador en dicha tabla con el mismo nombre y posterior a la última marca de bloque, se produce un error de identificador duplicado.

Inserción de una variable en la Tabla de Símbolos

```
...
resto_decl : resto_decl COMA IDENT  { TS_InsertaIDENT ($3) ; }
           | IDENT                  { TS_InsertaIDENT ($1) ; }
           ;
...
```

2. Al entrar en un bloque, se debe incluir en la tabla de símbolos una marca de inicio de bloque. En el caso de que el bloque corresponda al cuerpo de un subprograma, los parámetros formales (que se encuentran por encima de la marca en la tabla de símbolos) deberán ser introduciros de nuevo en la tabla como si fuesen variables locales al bloque.

Marca comienzo de bloque: Especificación YACC

```
 ...
 bloque : IBLOQ              { TS_InsertaMARCA(); }
          decl_variables
          decl_subprogramas
          sentencias
          FBLOQ ;
 ...
```

Marca comienzo de bloque: Especificación YACC

```
...
decl_subprog : cabec_subprog      { subProg= 1 ; }
               bloque             { subProg= 0 ; }
...
```

Donde `subProg` es una variable de tipo `boolean` que nos dice si el comienzo del bloque es el de un subprograma, en cuyo caso debe leer todos los parámetros formales situados justo antes de la marca de comienzo de bloque e insertarlos en la tabla de símbolos como si fuesen variables locales al bloque.

3. Al reconocer el tipo en una declaración de variables, será necesario asignar tipo a las variables de la tabla de símbolos que no lo tuviesen asignado (para lenguajes con el tipo después de los nombres de las variables) o bien asignar la variable temporal de tránsito `tipoTmp` con este tipo. Para lo anterior será necesario que el no terminal asociado con la descripción de tipo tenga como atributo sintetizado precisamente el registro descriptor de tipo dentro de la estructura atributos.

Tipo en Declaración de Variables

```
...
decl_var :   TIPOVAR     { tipoTmp= $1.tipo ; }
             resto_decl
             PCOMA
         ;
...
```

4. Al alcanzar el final de un bloque, se deben eliminar de la tabla de símbolos todas las entradas hasta la última marca de inicio de bloque, también incluida.

Marca fin de bloque: Especificación YACC

```
 ...
 bloque : IBLOQ              { TS_InsertaMARCA(); }
          decl_variables
          decl_subprogramas
          sentencias
          FBLOQ              { TS_VaciarENTRADAS(); }
 ...
```

5. Al reconocer un identificador que contenga el nombre de un subprograma, en la zona de declaraciones, se insertará en la tabla de símbolos una nueva entrada. El número de parámetros se pondrá inicialmente a cero.

Inserción de subprograma en la Tabla de Símbolos

```
...
cabecera_subprog : TIPOVAR IDENT ···   { TS_InsertaSUBPROG ($2) ; }
                 ;
...
```

6. Al reconocer un identificador en la lista de parámetros formales de la cabecera de un procedimiento o función, se insertará una entrada en la tabla de símbolos con el nombre y tipo del identificador. Además, será necesario incrementar el número de parámetros de dicho subprograma que se encontrará en una entrada anterior de la tabla de símbolos.

Inserción de Parámetros Formales en la Tabla de Símbolos

```
...
var_subprog : cuerpo_var_subprog
              COMA TIPOVAR IDENT      { TS_InsertaPARAMF ($4) ; }
            | TIPOVAR IDENT           { TS_InsertaPARAMF ($2) ; }
            |
            ;
...
```

F.5.4 Análisis Semántico

1. Todos los identificadores detectados deben aparecer en la tabla de símbolos.
2. En las sentencias de control de flujo (`if`, `while`, `for`, etc.) se debe comprobar que la expresión que forma la condición es de tipo booleano.
3. En las sentencias de asignación, el tipo de la variable en la parte izquierda debe coincidir con el tipo de la expresión en la parte derecha. Esto es también así para las asignaciones al contador en las sentencias tipo `for`.
4. En las expresiones compuestas de subexpresiones o términos relacionados con operadores, se debe comprobar que los tipos son concordantes y se debe sintetizar el tipo ascendentemente. Esto se realiza dando valor al campo `dtipo` de la pseudovariable `$$`.

Apéndice G

KAERU

G.1 Descripción

Se dispone de una herramienta software de libre distribución para facilitar la construcción de traductores basados en la utilización de LEX y YACC tal y como está planteado en las prácticas de la asignatura [Pére04].

Consiste en un entorno integrado implementado en Java que facilita la descripción del lenguaje en EBNF (práctica anterior) llevando la contabilidad de los símbolos no terminales y terminales que se hayan definido, así como su posterior transformación a BNF para una cómoda transcripción al formato de entrada de YACC. Posteriormente, es muy útil para las prácticas 3, 4 y 5 ya que contempla la posibilidad de determinar los tokens y símbolos no terminales, así como las acciones semánticas necesarias para las prácticas 4 y 5.

Toda la información referente a esta herramienta, manual de usuario, código fuente en Java y compilado (tanto para Linux como Windows) para su utilización se encuentra en la página web de la asignatura: `http://lsi.ugr.es/pl` en el apartado `Software`. Se recomienda leer primero el manual de usuario.

G.2 Ventajas de su Utilización

A la hora de escribir una especificación en EBNF son muy típicos los errores en los que no se definen la totalidad de símbolos no terminales. Kaeru dispone de un analizador que, mientras se escribe la especificación, informa si algún símbolo está o no definido. También informa del total de símbolos terminales y no terminales mostrándolos en una lista.

Otra ventaja es que la especificación EBNF servirá de origen para la confección de la especificación YACC que se debe escribir en BNF. Kaeru dispone de una opción que nos permite pasar de EBNF a BNF.

En una especificación YACC, el editor en vista de diseño muestra la lista de tokens que se han definido. La validación de la especificación identificará aquellos símbolos que no se hayan definido ni como tokens ni como símbolos no terminales.

Además, de cara a las prácticas 4 y 5, la especificación sintáctica dispone de una utilidad para mostrar u ocultar las acciones semánticas que se le añadan a las reglas sintácticas. De esta forma, es posible seguir teniendo la visión sintáctica que la abundancia de código en las acciones semánticas suele impedir, tal y

como se podrá ver en la vista de código de la aplicación.

Al tratarse de un desarrollo de software libre, han ido apareciendo ciertos fallos que aún no han sido resueltos. Dado que los símbolos en la especificación se acotan por «"y »", no es posible definir dichos caracteres en el lenguaje ya que Kaeru interpreta que entre dichos caracteres debe aparecer el nombre de un símbolo. Se dispone del código fuente hecho en Java de Kaeru para poder arreglar los fallos que se le encuentren, actividad que agradeceremos los profesores de la asignatura.

Apéndice H

SEFASGEN

Se dispone de una software de libre distribución para facilitar la construcción de un traductor en cuanto a las fases de análisis semántico y generación de código intermedio [Ferr05].

La principal ayuda se proporciona a la hora de implementar la tabla de símbolos de la práctica de análisis semántico. Sefasgen proporciona unas estructuras de datos y funciones para manejo de una tabla de símbolos acorde a los requerimientos de las prácticas, lo que supondría un ahorro a la hora de la implementación de la misma y, sobre todo, un entorno en el que es posible apreciar la evolución del proceso de compilación en cuanto a uso de la tabla, ya que se puede ver paso a paso la inserción, búsqueda y eliminación de entradas de la misma.

En cuanto a generación de código intermedio, posee un archivo en el que se establece el esquema de traducción. Cada grupo de prácticas tendrá asignado un lenguaje concreto que incorpora tipos de instrucciones diferentes. En dicho archivo se puede realizar la correspondencia a la hora de la traducción a un lenguaje de cuartetos. Además, se puede ir viendo paso a paso el código que se va generando, de cara a la comprobación del correcto funcionamiento de la práctica.

Toda la información referente a esta herramienta, manual de usuario y los binarios para su utilización se encuentra en la página web de la asignatura: `http://lsi.ugr.es/pl` en el apartado `Software`. Allí se encuentra un compilador de ejemplo basado en un lenguaje asignado. Se recomienda leer primero el manual de usuario.

Apéndice I

Lex y Yacc con Java

En este apéndice se proporciona un ejemplo de como usar las herramientas **jflex** y **byaccj** para crear procesadores de lenguajes en Java, incluyendo análisis léxico, sintáctico y semántico. El ejemplo es el típico de la calculadora (evalua expresiones aritméticas sencillas), aumentada con la posibilidad de asociar valores a identificadores (variables). Toda esta información está disponible en el siguiente enlace:

`http://lsi.ugr.es/~curena/doce/pl/lyj`

I.1 Descarga de herramientas

Las herramientas *jflex/byaccj* son de libre disposición, con código abierto. Se puede descargar el código fuente y algunas versiones compiladas en estos sitios web:

- **jflex:** En `jflex.de` está la web del proyecto. Hay paquetes ya creados para casi todas las distribuciones de linux. Están disponibles para ubuntu, fedora, gentoo y debian.
- **byacc/j:** En `byaccj.sourceforge.net` está la web del proyecto y es posible descargarla desde el propio directorio en sourceforge: `http://sourceforge.net/projects/byaccj`.

(comprobados en Febrero de 2008)

I.2 Analizador Léxico (clase `YyParser`)

La herramienta *jflex* tiene una funcionalidad similar a flex o lex. En concreto, *jflex* sirve para producir código a que realiza análisis léxico, a partir de una especificación de las expresiones regulares que definen los tokens, y de las acciones a llevar a cabo al reconocerlos.

El programa `jflex` produce código Java a partir de esta especificación, en concreto, produce un archivo de nombre Yylex.java, con la definicion de la clase `Yylex`, que encapsula el código de análisis léxico

En el ejemplo, el archivo fuente para *jflex* se llama `prueba.l`, y aparece aquí:

Listing I.1: prueba.l

```
%%
%byaccj
%{
  private Parser yyparser ;

  // constructor del analizador léxico que acepta como parámetro el
  // lector de caracteres a usar

  public Yylex(java.io.Reader r, Parser p )
  {
    this(r);
    linea_actual = 1 ;
    yyparser = p  ;
  }

  // guarda el número de linea actual
  //
  private int linea_actual ;

  // devuelve el numero de linea donde está el último caracter leido
  //
  public int lineaActual()
  {
     return linea_actual ;
  }
%}
%%
"="                   {  return Parser.ASIG ; }
"("                   {  return Parser.ABR_PARENT ; }
")"                   {  return Parser.CER_PARENT ; }
";"                   {  return Parser.PYC ; }
"+"|"-"               {  yyparser.yylval = new ParserVal( new Character(yytext().charAt(0)) ) ;
                         return Parser.OP_MAS_MENOS ; }
"*"|"/"               {  yyparser.yylval = new ParserVal( new Character(yytext().charAt(0)) ) ;
                         return Parser.OP_MULT_DIV ; }
[0-9]+                {  yyparser.yylval = new ParserVal( new Integer(yytext()) );
                         return Parser.CONSTANTE ; }
[0-9]+"."[0-9]+ ?     {  yyparser.yylval = new ParserVal( new Double(yytext()) );
                         return Parser.CONSTANTE ; }
[a-zA-Z][a-zA-Z0-9_]* {  yyparser.yylval = new ParserVal( (Object) yytext() );
                         return Parser.IDENTIFICADOR ;  }
\n                    {  // lleva la cuenta de lineas
                         linea_actual ++ ; }
[ \t]+                {  // accion vacia: se ignoran los espacios y tabuladores }
.                     {  // token desconocido: se produce un mensaje de error
                         yyparser.yyerror("el(los) carácter(es) '"+yytext()+
                                          "' no forma(n) ningún token conocido"); }
```

I.3 Analizador Sintáctico

La herramienta *byacc/j* es una adaptación de *byacc* para crear analizadores sintácticos en Java. Tiene exactamente la misma funcionalidad que *byacc*, aunque las acciones se escriben, por supuesto, en Java.

El código se genera en el archivo `Parser.java`, que contiene la declaracion de la clase `Parser` (una ventaja frente a C es que se pueden instanciar más de un parser). Esta clase contiene las tablas de análisis y el código correspondiente a las acciones necesarias para análisis semántico y generación de código o traducción. En el ejemplo, las acciones son llamadas a métodos de la clase Acc, que, por simplicidad, se ha incluido en un archivo independiente (ver el listado más abajo).

El tipo de datos para la pila de atributos sintetizados es `ParserVal`, una clase definida en el archivo `ParserVal.java`, archivo que también genera *byaccj* al ejecutarse. En el ejemplo, se usa exclusivamente la variable de instancia `obj` de `ParserVal`. Puesto que `obj` es de tipo `Object`, en tiempo de

ejecución podemos usar `obj` para almacenar una referencia a cualquier tipo de objeto, con lo cual los atributos sintetizados pueden ser de cualquier tipo.

En el ejemplo, es el analizador sintáctico el que crea una instancia del analizador léxico (`analiex`) y la usa cuando requiere un nuevo token.

Al igual que *byacc*, *byaccj* no informa de los tokens que se esperan cuando ocurre un error sintáctico. En el ejemplo, se ha implementado la rutina de error de forma que se haga ese informe, leyendo las tablas de análisis del analizador sintáctico.

Aquí se incluye el contenido de `prueba.y`, el fuente que se proporciona a byaccj.

Listing I.2: prueba.y

```
%{
  import java.io.*;
%}

// lista de tokens por orden de prioridad

%token CONSTANTE      // constante (entero o flotante)
%token ABR_PARENT     // abrir paréntesis
%token CER_PARENT     // cerrar paréntesis
%token ASIG           // simbolo usado para la asignación
%token IDENTIFICADOR
%token PYC            // punto y coma
%left  OP_MAS_MENOS   // operadores binarios de menos prioridad (o unarios)
%left  OP_MULT_DIV    // operadores binarios de mas prioridad
%token PEPE           // token para hacer pruebas

%%
lista_sent :  lista_sent sentencia
           | ;
sentencia  :  expr PYC                    {  System.out.println($1.obj.toString());  }
           |  IDENTIFICADOR ASIG expr PYC {  Acc.Asignacion($1.obj,$3.obj) ; }
           |  PEPE PYC
           |  error PYC
;
expr       :  CONSTANTE                  {  $$.obj = $1.obj ; }
           |  IDENTIFICADOR              {  $$.obj = Acc.LeerValorVariable($1.obj) ; }
           |  expr OP_MAS_MENOS expr     {  $$.obj = Acc.Operador($2.obj,$1.obj,$3.obj); }
           |  expr OP_MULT_DIV expr      {  $$.obj = Acc.Operador($2.obj,$1.obj,$3.obj); }
           |  OP_MAS_MENOS expr          {  $$.obj = Acc.Operador($1.obj,$2.obj);        }
           |  ABR_PARENT expr CER_PARENT {  $$.obj = $2.obj ; }
;
%%
// referencia al analizador léxico
//
private Yylex analex ;

// constructor: crea el analizador sintactico
// invoca la creación del analizador lexico (lexer)
//
public Parser(Reader r)
{
   analex = new Yylex(r, this) ;
}

// esta función se invoca por el analizador cuando necesita el
// siguiente token del analizador léxico
//
private int yylex ()
{
  int yyl_return = -1;
  try
  {  yylval = new ParserVal(0);
     yyl_return = analex.yylex();
  }
  catch (IOException e)
  {  System.err.println("error␣de␣E/S:"+e);
  }
  return yyl_return;
```

```
}

// invocada cuando se produce un error sintáctico
//
public void yyerror (String descripcion, int yystate, int token)
{
   System.err.println ("Error en línea "+Integer.toString(analex.lineaActual())+
                       " : "+descripcion);
   System.err.println ("Token leído : "+yyname[token]);
   System.err.print("Token(s) que se esperaba(n) : ");

   String  nombresTokens = "" ;

   int yyn ;

     // añadir en 'nombresTokens' los tokens que permitirian desplazar
     for( yychar = 0 ; yychar < YYMAXTOKEN ; yychar++ )
     {  yyn = yysindex[yystate] ;
        if ((yyn != 0) && (yyn += yychar) >= 0 &&
           yyn <= YYTABLESIZE && yycheck[yyn] == yychar)
        {   nombresTokens += yyname[yychar] + " ";
        }
     }

     // añadir tokens que permitirian reducir
     for( yychar = 0 ; yychar < YYMAXTOKEN ; yychar++ )
     {   yyn = yyrindex[yystate] ;
         if ((yyn !=0 ) && (yyn += yychar) >= 0 &&
            yyn <= YYTABLESIZE && yycheck[yyn] == yychar)
         {  nombresTokens += yyname[yychar] + " " ;
         }
     }

    System.err.println(nombresTokens);
  }

  public void yyerror (String descripcion)
  {
     System.err.println ("Error en línea "+Integer.toString(analex.lineaActual())+
                         " : "+descripcion);
  }
```

I.4 Acciones del Analizador (clase Acc)

En el ejemplo, y por simplicidad, las acciones correspondientes al analizador se han incluido en la clase Acc, en el archivo `Acc.java`, que se incluye a continuación. La evaluación de las expresiones se hace generando como atributos sintetizados el valor de las mismas, que puede ser entero (el atributo es una instancia de `Integer`) o real (el atributo es una instancia de `Double`)

Listing I.3: clase Acc

```
// fuente Java para las acciones semánticas de una calculadora sencilla

import java.util.Hashtable ;

class Acc
{
  // tabla con las variables definidas y sus valores
  private static Hashtable variables = new Hashtable() ;
             // documentacion sobre 'Hashtable' aqui:
             // http://java.sun.com/j2se/1.4.2/docs/api/java/util/Hashtable.html

  // cambia o inicializa el valor de una variable
  // (ident debe ser String)
  public static void Asignacion(Object ident, Object valor)
  {
       variables.put((String)ident,valor);
  }
```

```
// devuelve el valor de una variable (0 entero si no existe, y da mensaje)
// (ident debe ser String)
public static Object LeerValorVariable(Object ident)
{
    if ( variables.containsKey((String)ident) )
       return variables.get((String)ident) ;
    else
    {
       System.out.println("error: la variable '"+ident+"' no existe (se usa el valor 0)");
       return new Integer(0) ;
    }
}

// convierte el objeto 'obj' a double
// 'obj' puede ser 'Double' o 'Integer'
//
public static double ValorReal( Object obj )
{
    if ( obj.getClass() == Double.class )
        return ((Double) obj).doubleValue() ;
    else // se asume obj.getClass() == Integer.class
        return ((Integer) obj).doubleValue() ;
}

// evalua un operador binario aplicado a dos operandos, devuelve el resultado
// (los operandos pueden ser de tipo Integer o Double)
//
public static Object Operador( Object operador, Object operando1, Object operando2 )
{
    if ( operando1.getClass() == Double.class || operando2.getClass() == Double.class )
    {
        double op1 = ValorReal(operando1) ,
               op2 = ValorReal(operando2) ;

        switch ( ( (Character) operador ).charValue() )
        {
           case '+' : return op1+op2 ;
           case '-' : return op1-op2 ;
           case '*' : return op1*op2 ;
           case '/' : return op1/op2 ;
           default  : System.out.println("error interno: operador binario desconocido");
                      return 0 ;
        }
    }
    else
    {
        int op1 = ((Integer)operando1).intValue() ,
            op2 = ((Integer)operando2).intValue() ;

        switch ( ( (Character) operador ).charValue() )
        {
           case '+' : return op1+op2 ;
           case '-' : return op1-op2 ;
           case '*' : return op1*op2 ;
           case '/' : return op1/op2 ;
           default  : System.out.println("error interno: operador binario desconocido");
                      return 0 ;
        }
    }
}

// evalua un operador unario aplicado a un operando, devuelve el resultado
// (el operando puede ser de tipo Integer o Double)
//
public static Object Operador( Object operador, Object operando )
{
    if ( operando.getClass() == Double.class )
    {
        switch ( ( (Character) operador ).charValue() )
        {
           case '+' : return operando ;
           case '-' : return - ValorReal(operando) ;
           default  : System.out.println("error interno: operador unario desconocido.");
                      return 0.0 ;
        }
```

```
      }
      else  // se asume que: operando.getClass() == Integer.class
      {
          switch ( ( (Character) operador ).charValue()  )
          {
             case '+' : return operando ;
             case '-' : return - ((Integer)operando).intValue() ;
             default  : System.out.println("error␣interno:␣operador␣unario␣desconocido");
                        return 0 ;
          }
      }
  }

} // class Acc
```

I.5 Método Principal (clase `Main`)

Esta clase contiene el método principal del programa analizador de ejemplo. El método crea el objeto (de la clase `Reader`) usado para leer caracteres (un archivo o la entrada estándard), crea una instancia del parser (se llama `analizador`) y finalmente invoca el método `yyparse` en dicha instancia para proceder al análisis sintactico.

Listing I.4: clase Main

```
import java.io.* ;

class Main
{

  private static Reader AbrirLector(String args[])
  {
    Reader lector = null ;

    if ( args.length > 0 )
    {
       try
       {
          lector = new FileReader(args[0]) ;
       }
       catch( IOException exc )
       {
          System.err.println("imposible␣abrir␣archivo␣'"+args[0]+"'");
          System.err.println("causa:␣"+exc.getMessage()) ;
          System.exit(1) ;
       }

       System.out.println("leyendo␣archivo␣'"+args[0]+"'");
    }
    else
    {
       lector = new InputStreamReader(System.in) ;
       System.out.println("leyendo␣entrada␣estándard␣(terminar␣con␣ctrl-d)");
    }

    return lector ;
  }

  /*************************************************************************/

  public static void main(String args[]) throws IOException
  {
      Parser analizador = new Parser(AbrirLector(args)) ;
      analizador.yyparse();
  }
}
```

I.6 Compilación y Ejecución (archivo `makefile`)

El proceso de ejecutar *jflex*, *byaccj*, *javac* y *java* para compilar y ejecutar el analizador puede ser tedioso. Este `makefile` simplifica dicho proceso, que puede ser lanzado escribiendo simplemente `make`. El archivo, además, sirve como documentación de las dependencias que existen entre los distintos archivos.

Este archivo `makefile` contiene, además, una sentencia para reemplazar una llamada al método para mostrar el mensaje de error `Parser.yyerror` en `Parser.java`. El motivo es que es necesario insertar en dicha llamada, como parámetros, el estado del autómata del analizador sintáctico, así como el siguiente token a leer, de forma que sea posible informar correctamente de los tokens que se esperan (véase el fuente del analizador sintáctico).

La sentencia `make` compila (si es necesario) y ejecuta el programa, usando como entrada el archivo `entrada.txt`. El analizador se puede ejecutar directamente (leyendo de la entrada estándar), usando la orden `java Main`.

Listing I.5: archivo makefile

```
JFLEX  = jflex
BYACCJ = ../lyj/byaccj -J # cambiar según proceda
JAVAC  = javac

.SUFFIXES:
.PHONY: ejecuta

ejecuta: Main.class entrada.txt
        java Main entrada.txt

Main.class: Ylex.java Parser.java ParserVal.java Main.java Acc.java
        rm -f *.class
        $(JAVAC) Main.java

Ylex.java: prueba.l
        $(JFLEX) prueba.l

Parser.java ParserVal.java: prueba.y
        $(BYACCJ) prueba.y
        sed s/'yyerror("syntax␣error")'/'yyerror("error␣sintactico",yystate,yychar)'/ \
        Parser.java >tmp.java
        mv tmp.java Parser.java

limpia:
        rm -f *~ *.class Ylex.java Parser.java ParserVal.java
```

I.7 Consideraciones sobre las Prácticas usando Java

Este ejemplo es útil viéndolo en global como un traductor con todas sus etapas, sin embargo, a la hora de realizar la prueba de la práctica de análisis de léxico se tiene que obtener un ejecutable que muestre los lexemas, código del token y, en su caso, el atributo correspondiente.

En ese caso, dado que aún no se usaría `byaccj`, habría que definir los códigos de los tokens y, en su caso, los atributos y mostrarlos en pantalla para comprobar que la fase de análisis léxico funciona correctamente. Para las fases de análisis sintáctico, semántico y de generación de código intermedio, debe tomarse como ejemplo los anteriores dado que integran toda la funcionalidad del traductor.

Bajo estas premisas, el ejemplo de análisis léxico quedaría así:

Listing I.6: prueba-sololexico.l

```
// fuente 'jflex' de pruebas.
%%
%byaccj
%{
  // guarda el número de linea actual
  //
  private int linea_actual = 1 ;

  // devuelve el numero de linea donde está el último carácter leído
  //
  public int lineaActual()
  {
     return linea_actual ;
  }
  // informa de que se ha reconocido un token
  //
  void EscribeToken(int codigo, String nombre)
  {
      String strCod = (new Integer(codigo)).toString() ;

      System.out.println("reconocido␣token␣con␣código␣"+strCod+
                         "␣("+nombre+"),␣lexema␣=␣'"+yytext()+"'") ;
  }
  // codigos de tokens
  //
  private final static int ASIG           = 257 ;
  private final static int ABR_PARENT     = 258 ;
  private final static int CER_PARENT     = 259 ;
  private final static int PYC            = 260 ;
  private final static int OP_MAS_MENOS   = 261 ;
  private final static int OP_MULT_DIV    = 262 ;
  private final static int CONSTANTE      = 263 ;
  private final static int IDENTIFICADOR  = 264 ;
%}
%%
"="     { EscribeToken(ASIG,"ASIG");
          return ASIG ;
        }
"("     { EscribeToken(ABR_PARENT,"ABR_PARENT");
          return ABR_PARENT ;
        }
")"     { EscribeToken(CER_PARENT,"CER_PARENT");
          return CER_PARENT ;
        }
";"     { EscribeToken(PYC,"PYC");
          return PYC ;
        }
"+"|"-" {  EscribeToken(OP_MAS_MENOS,"OP_MAS_MENOS");
           return OP_MAS_MENOS ;
        }
"*"|"/" {  EscribeToken(OP_MULT_DIV,"OP_MULT_DIV");
           return OP_MULT_DIV ;
        }
[0-9]+  {  EscribeToken(CONSTANTE,"CONSTANTE");
           return CONSTANTE ;
        }
[0-9]+"."[0-9]+ ?
        {  EscribeToken(CONSTANTE,"CONSTANTE");
           return CONSTANTE ;
        }
[a-zA-Z][a-zA-Z0-9_]*
        {
           EscribeToken(IDENTIFICADOR,"IDENTIFICADOR");
           return IDENTIFICADOR ;
        }
\n      { // lleva la cuenta de lineas
          linea_actual ++ ;
        }
[ \t]+  {  /** accion vacia: se ignoran los espacios y tabuladores **/
        }
.       {  /** token desconocido: se produce un mensaje de error **/
           System.err.println("el(los)␣carácter(es)␣'"+yytext()+
                              "'␣no␣forma(n)␣ningún␣token␣conocido");
        }
```

De esta forma, el fichero `main.java` sería el siguiente

Listing I.7: clase Main (solo Léxico)

```
import java.io.* ;

class Main
{

  private static Reader AbrirLector(String args[])
  {
    Reader lector = null ;

    if ( args.length > 0 )
    {
       try
       {
          lector = new FileReader(args[0]) ;
       }
       catch( IOException exc )
       {
          System.err.println("imposible␣abrir␣archivo␣'"+args[0]+"'");
          System.err.println("causa:␣"+exc.getMessage()) ;
          System.exit(1) ;
       }

       System.out.println("leyendo␣archivo␣'"+args[0]+"'");
    }
    else
    {
       lector = new InputStreamReader(System.in) ;
       System.out.println("leyendo␣entrada␣estandard␣(terminar␣con␣ctrl-d)");
    }

    return lector ;
  }

  /*************************************************************************/

  public static void main(String args[]) throws IOException
  {
      Yylex analizador_lexico = new Yylex(AbrirLector(args)) ;

      while ( analizador_lexico.yylex() != 0 ) {}
  }

}
```

Por último, el `makefile` quedaría así:

Listing I.8: archivo makefile (solo Léxico)

```
JFLEX  = jflex
JAVAC  = javac
.SUFFIXES:
.PHONY: ejecuta

ejecuta: Main.class entrada.txt
        java Main entrada.txt

Main.class: Yylex.java  Main.java
        rm -f *.class
        $(JAVAC) Main.java

Yylex.java: prueba.l
        $(JFLEX) prueba.l

limpia:
        rm -f *~ *.class Yylex.java

SRCS = prueba.l entrada.txt Main.java makefile

lj.tgz: $(SRCS)
        tar czvf lj.tgz $(SRCS)
```

Bibliografía

[Biso06] Bison, *Bison documentation*, http://www.gnu.org/software/bison/manual, 2006.

[Ferr05] S. Ferrer, J. Revelles, *SEFASGEN: Software para la Enseñanza de las Fases de Análisis Semántico y Generación de Código*, Tech. Report PFC-2005, Dpt. Lenguajes y Sistemas Informáticos, 2005.

[Flex06] Flex, *Flex documentation*, http://www.gnu.org/software/flex/manual, 2006.

[Lesk75] M.E. Lesk, E. Schmidth, *UNIX Programmer's Manual 2*, vol. 2, UNIX, 1975.

[Levi92] J.R. Levine, T. Manson, D. Brown, *Lex & Yacc*, O'Reilly & Associates, Inc., 1992.

[Lex06] Lex, *Lex documentation*, http://dinosaur.compilertools.net/#lex, 2006.

[Pére04] G. Pérez, S. Sánchez, J. Revelles, *KAERU: Entorno Integrado para la Generación Semiautomática de Traductores*, Tech. Report PFC-2004, Dpt. Lenguajes y Sistemas Informáticos, 2004.

[Yacc06] Yacc, *Yacc documentation*, http://dinosaur.compilertools.net/#yacc, 2006.