Procesamiento de Lenguajes (PL)
Curso 2019/2020

Práctica 4: traductor ascendente usando `bison/flex`

Fecha y método de entrega

La práctica debe realizarse de forma individual, como las demás prácticas de la asignatura, y debe entregarse a través del servidor de prácticas del DLSI **antes de las 23:59 del 3 de mayo de 2020**. Los ficheros fuente ("`plp4.l`", "`plp4.y`", "`comun.h`", "`TablaSimbolos.h`", "`TablaSimbolos.cc`", "`Makefile`" y aquellos que se añadan) se comprimirán en un fichero llamado "`plp4.tgz`", sin directorios.

Al servidor de prácticas del DLSI se puede acceder desde la web del DLSI (http://www.dlsi.ua.es), en el apartado "Entrega de prácticas". Una vez en el servidor, se debe elegir la asignatura PL y seguir las instrucciones.

Descripción de la práctica

La práctica consiste en implementar un traductor ascendente utilizando las herramientas `bison` y `flex`, versiones modernas de `yacc` y `lex`, para el mismo proceso de traducción de la práctica 3.

Gramática

La gramática que debe utilizarse como base para diseñar el traductor es la siguiente:

X	$\longrightarrow$	*S*
S	$\longrightarrow$	**class id lbra** *M* **rbra**
M	$\longrightarrow$	*M SF*
M	$\longrightarrow$	ϵ
SF	$\longrightarrow$	*S*
SF	$\longrightarrow$	*Fun*
Fun	$\longrightarrow$	**fun id** *A* **lbra** *M Cod* **rbra**
A	$\longrightarrow$	*A* **pyc** *DV*
A	$\longrightarrow$	*DV*
DV	$\longrightarrow$	*Tipo* **id**
Tipo	$\longrightarrow$	**int**
Tipo	$\longrightarrow$	**float**
Cod	$\longrightarrow$	*Cod* **pyc** *I*
Cod	$\longrightarrow$	*I*
I	$\longrightarrow$	*DV*
I	$\longrightarrow$	**lbra** *Cod* **rbra**
I	$\longrightarrow$	**id asig** *Expr*
I	$\longrightarrow$	**if** *Expr* **dosp** *I Ip*
Ip	$\longrightarrow$	**else** *I* **fi**
Ip	$\longrightarrow$	**fi**
I	$\longrightarrow$	**print** *Expr*
Expr	$\longrightarrow$	*E* **oprel** *E*
Expr	$\longrightarrow$	*E*
E	$\longrightarrow$	*E* **opas** *T*
E	$\longrightarrow$	*T*
T	$\longrightarrow$	*T* **opmul** *F*
T	$\longrightarrow$	*F*
F	$\longrightarrow$	**numentero**
F	$\longrightarrow$	**numreal**
F	$\longrightarrow$	**id**
F	$\longrightarrow$	**pari** *Expr* **pard**

Observa que, para facilitar el diseño del traductor (especialmente en lo relacionado con los atributos heredados y su implementación en traductores ascendentes), hay muchos no terminales que presentan recursividad por la izquierda, que **no** debes eliminar. Algunas reglas de la gramática se podrían simplificar, pero al hacerlo probablemente

se producirían conflictos reducción/reducción al introducir marcadores, por eso es recomendable no modificar la gramática (está basada en una implementación que funciona bien).

Mensajes de error

Los errores léxicos y semánticos deben generar exactamente el mismo mensaje que en la práctica 3, con el mismo formato. En el caso de los mensajes de error sintáctico, el mensaje de error debe ser simplemente:

```
Error sintactico (fila,columna): en 'lexema'
```

Esto se debe a que las tablas generadas por `bison` son difíciles de interpretar y por tanto, por simplificar, no es necesario mostrar los tokens que se esperaban en el lugar del token incorrecto.

En la página web de la asignatura se dejará un fichero con funciones para emitir mensajes de error léxico, sintáctico y semántico.

Notas técnicas

1. Los programas `yacc` (`bison`) y `lex` (`flex`), a partir de una especificación léxica y un ETDS, generan un traductor ascendente basado en un analizador sintáctico LALR(1), escrito en C/C++. Junto con este enunciado se publicará una hoja técnica acerca de estas herramientas, con un ejemplo práctico en el que basarse para hacer la práctica.

   Este ejemplo contiene 3 ficheros (y además un script para compilarlo, y un ejemplo):

   - `ejemplo.y`: especificación del traductor (ETDS), junto con funciones para emitir mensajes de error, y la función `main`
   - `ejemplo.l`: especificación de los *tokens* del lenguaje fuente (los `#define` con los números de *token* se generan a partir de `ejemplo.y` en el fichero `ejemplo.tab.h`), junto con una función y variables auxiliares para asociar a cada token su fila y su columna en el programa fuente. También incluye reglas para la gestión de comentarios de una línea o de varias líneas.[1]
   - `comun.h`: definiciones comunes para ambos ficheros:
     - Tipo de los atributos (`YYSTYPE`), comunes a todos los símbolos terminales y no terminales (aunque según el símbolo, algunos atributos no tendrán valor). En este ejemplo, los atributos `lexema`, `nlin` y `ncol` se usan con los terminales, y `tipo` y `cod` con los no terminales (aunque no todos los no terminales usan `tipo`).[2]
     - Tipos de errores y función para emitir mensajes de error.

2. Como en las prácticas anteriores, se publicará un autocorrector para facilitar la tarea de depurar la práctica.

3. El proceso de traducción es el mismo que en la práctica 3, para lo que es necesario utilizar una tabla de símbolos (con básicamente dos funciones, `anyadir` y `buscar`). En la web de la asignatura se publicará el código fuente de una clase para manejar tablas de símbolos con ámbitos anidados, similar a la misma clase en Java de la práctica 3. Como en la práctica 3, se recomienda usar una variable global para guardar la tabla de símbolos del ámbito actual:

   ```
   TablaSimbolos *tsa=new TablaSimbolos(NULL);  // inicialización
   ```

   Cada vez que se abre un nuevo ámbito, se crea una nueva tabla:

   ```
   tsa = new TablaSimbolos(tsa);
   ```

   Cuando el ámbito termina, se recupera la tabla de símbolos del ámbito anterior (el ámbito *padre*):[3]

   ```
   tsa = tsa->padre;  // no libero memoria
   ```

[1] Ten en cuenta que el lenguaje fuente de la práctica solo admite un tipo de comentarios, no ambos.

[2] En la época en la que se desarrolló `yacc` la memoria era escasa, por lo que se usaban `union` para `YYSTYPE`, y `yacc` está preparado para ello, pero hoy en día no tiene mucho sentido, y resta flexibilidad, por eso este ejemplo usa un `struct`.

[3] No es necesario liberar memoria en esta práctica.

4. Aunque no es imprescindible, se pueden añadir más ficheros fuente a la práctica, para lo que será necesario modificar el `Makefile` que se publicará en la web de la asignatura. Por supuesto, esos ficheros deben incluirse en el fichero comprimido `plp4.tgz` que se entregue.

5. Para comprobar que después de un programa correcto no aparecen más tokens en el fichero, es decir, que el siguiente token es el del final del fichero, es necesario hacer una comprobación en la acción semántica situada al final de la regla:[4]

$$X \longrightarrow S$$

   En dicha acción semántica se debe llamar directamente al analizador léxico:

```
int token = yylex();
```

   Si el valor de `token` es 0, el fichero termina correctamente. Si es cualquier otro valor, hay que generar un error sintáctico con la llamada "`yyerror("")`"

[4] Véase el ejemplo de la hoja técnica.