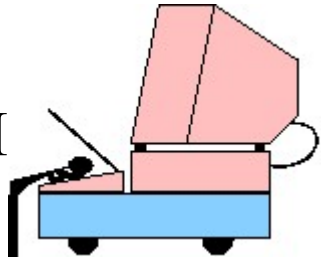
Fundamentos de Ordenadores I



Práctica 1

Entrega: hasta el 26 de abril de 2004

Entrega: A través del enlace http://www.it.uc3m.es/luis/fo1/p1/submit.html

---

Introducción a JLex y CUP

Tutoriales

Antes de empezar a realizar la práctica se recomienda consultar los manuales de CUP y JLex disponibles a través de la página Web de la asignatura o bien estos pequeños resúmenes de CUP y JLex

Instalación de CUP y JLex

La instalación de CUP y JLex son muy sencillas. En el caso de CUP, lo único que hay que hacer es bajárselo (la versión recomendada es la 0.10k) del sitio Web de CUP o de la copia en el sitio Web de la asignatura.

A continuación crearemos un directorio donde queramos tener instalado CUP. Descomprimiremos la distribución de CUP que nos hemos descargado en ese directorio. Situados en el directorio donde hemos descomprimido la distribución de CUP ejecutaremos el siguiente comando:

```
javac java_cup/*.java java_cup/runtime/*.java
```

Si la compilación se realiza sin errores tendremos CUP correctamente instalado.

La instalación de JLex es todavía más sencilla. Lo único que tenemos que hacer es bajarnos el único fichero Java que lo compone (la versión recomendada es la 1.2.6), bien del sitio Web de JLex o bien de la copia en la página Web de la asignatura.

A continuación crearemos un directorio donde queramos tener instalado JLex y copiaremos en él el fichero `Main.java` que nos hemos bajado. Compilaremos el fichero Java (`javac Main.java`). Si la compilación no produce errores hemos terminado con la instalación de JLex.

Compilación de clases Java en `packages`

Al ejecutar `java` o `javac` para ejecutar o compilar clases Java, estas aplicaciones buscan las clases utilizadas por la clase que estamos ejecutando o compilando en ciertos directorios. Estos directorios se definen por medio de lo que se llama el classpath. Hay 2 formas de definir el classpath: por medio de una variable de entorno y mediante la opción `-classpath` de `java` y `javac`. Una variable de entorno es una variable que el sistema operativo le pasa a un programa al ser ejecutado; no confundir con una variable de un programa que es creada dentro del propio programa. Para ver como definir el classpath consultar la documentación del SDK que se esté utilizando.

Cuando la clase que se está buscando pertenece a un paquete, se supone que los paquetes son subdirectorios de alguno de los directorios indicados en el classpath. Así por ejemplo, si el classpath define los directorios donde buscar las clases `/usr/java/` y `/home/fo1/class/` y se busca la clase `Yylex` del paquete `Compilador.Lexer`, se buscará si existe un fichero de nombre `Yylex.class` en los directorios `/usr/java/Compilador/Lexer/` y `/home/fo1/class/Compilador/Lexer/`.

Cuando se compila una clase Java con `javac` por defecto el fichero en formato `class` se deja en el directorio en el que está la clase Java. Esto ocasiona problemas si la clase está en un paquete, ya que si posteriormente compilamos otra clase que usa la primera y está en el mismo directorio , y suponiendo que el directorio donde estamos compilando está en el classpath, la primera clase se buscaría en un subdirectorio del directorio donde estamos compilando y no se encontraría.

Por este motivo y porque se considera que es más modular separar los ficheros .class de las clases Java fuente, es **muy recomendable** que cuando se desarrollen proyectos de tamaño medio/grande de programación en Java, en los que se utilicen paquetes, se estructure el proyecto de programación en un directorio que contenga dos subdirectorios `java` y `class` que contengan los ficheros Java y los ficheros en formato .class respectivamente.

El directorio `java`, a su vez contendría subdirectorios que reflejasen la estructura de paquetes utilizada. Así, en el ejemplo anterior, debería existir un directorio `java/Compilador/Lexer/`, que contendría el fichero `Yylex.java`.

Para compilar las clases Java si tenemos esta estructura de directorios, debemos de utilizar la opción `-d` de `javac`, que nos permite identificar el directorio donde queremos dejar los ficheros .class. Siguiendo con el ejemplo, situados en el directorio `java/Compilador/Lexer/`, para compilar la clase `Yylex.java` ejecutaríamos lo siguiente:

```
javac -d ../../../class Yylex.java
```

El directorio indicado en la opción `-d` deberá haber sido creado previamente. Sin embargo, cuando compilamos con la opción `-d`, `javac` se encarga de crear automáticamente los directorios correspondientes a los paquetes. En este ejemplo, `javac` crearía el directorio `class/Compilador/Lexer/` automáticamente, en el caso de que éste no existiese previamente.

Debemos tener en cuenta que si para compilar la clase java es necesario utilizar alguna de las clases java que hemos compilado anteriormente, debemos indicar el directorio `class` en el classpath:

```
javac -d ../../../class -classpath ../../../class Yylex.java
```

Ejemplo JLex

Nota: para probar este ejemplo es necesario haber instalado previamente CUP, ya que el analizador léxico utiliza clases Java que forman parte de la distribución de CUP.

En su cuenta cree un directorio `EjemplosFOI`. Dentro de `EjemplosFOI` cree un directorio `java` y otro `class`. Dentro de `EjemplosFOI/java` cree un directorio `Lexer` y otro `Parser`.

Copie al directorio `EjemplosFOI/java/Lexer` este pequeño ejemplo de fichero JLex y esta clase java que contiene un método `main` con el que probar el analizador léxico que va a obtener.

Abra el fichero JLex e intente entender lo que hace. Observando el fichero JLex, vemos que en las acciones que generan los objetos `Symbol` que representan los tokens se ha añadido una sentencia que imprime en pantalla el token que se ha reconocido, lo que nos permitirá comprobar el comportamiento del analizador léxico generado.

Para obtener el analizador léxico ejecutaremos:

```
java JLex.Main Yylex
```

(deberá ser posible encontrar el paquete JLex en el classpath)

Se obtendrá el analizador léxico que vamos a probar.

Ahora debemos compilar las clases Java `Yylex.java` y `LexerMain.java` siguiendo las indicaciones que se han explicado en el apartado anterior. Para compilar la clase `Yylex.java` es necesario que la distribución de CUP esté accesible con el classpath.

Para probar el analizador léxico ejecutaremos:

```
java LexerMain
```

(se deberá utilizar el classpath adecuado: se usan clases de la distribución de CUP y se deberá poder acceder a la clase `Lexer.Yylex`)

A través de teclado podemos introducir texto (solamente se reconocen los dígitos y los símbolos "(", ")", ";", "+" y "*"). Cada vez que pulsemos el retorno de carro se imprimirá en pantalla la cadena de tokens equivalente la texto introducido por teclado. Para terminar pulsaremos simultáneamente las teclas "Ctrl" y "D".

Ejemplo CUP (con JLex)

Ahora vamos a probar un ejemplo que utiliza un fichero en formato CUP junto con el ejemplo JLex anterior para hacer una sencilla calculadora.

Copie al directorio `EjemplosFOI/java/Parser` este pequeño ejemplo de fichero CUP. Abrálo e intente entender lo que hace.

Para obtener el analizador sintáctico ejecutaremos:

```
java java_cup.Main minimal.cup
```

(deberá ser posible encontrar el paquete CUP en el classpath)

Se obtendrá el analizador sintáctico que vamos a probar.

Ahora debemos compilar las clases Java `parser.java` y `sym.java`, obtenidas al generar el analizador sintáctico. Para compilar estas clases es necesario que la distribución de CUP esté accesible con el classpath.

Para probar el analizador sintáctico ejecutaremos:

```
java Parser.parser
```

(se deberá utilizar el classpath adecuado: se usan clases de la distribución de CUP y se deberá añadir el directorio `EjemplosFOI/class/` al classpath para poder acceder a los paquetes `Lexer` y `Parser`)

Al igual que en el caso anterior, a través del teclado podemos introducir texto. Pruebe, por ejemplo, a introducir el siguiente texto:

```
(3 + 4) * 5;
```

Para terminar pulsaremos simultáneamente las teclas "Ctrl" y "D".

Definición del lenguaje de programación a traducir

Introducción

La práctica obligatoria de Fundamentos de Ordenadores I consiste en desarrollar un sencillo compilador que traduzca programas escritos en un pequeño lenguaje fuente a Java. Este curso académico se ha escogido un subconjuto mínimo de un lenguaje de simulación, utilizado para describir circuitos electrónicos digitales, llamado VHDL.

Simuladores y lenguajes de simulación

Los lenguajes de simulación son lenguajes que se utilizan para producir una representación de un determinado sistema. Esta representación podrá después ser "simulada" con un simulador para el lenguaje de simulación en cuestión. Un simulador es un programa que toma como entrada:

- La representación del sistema a simular.
- El tiempo de simulación. Especifica el intervalo de tiempo para el cual se desea simular el sistema.
- El valor que toman las entradas del sistema a simular a lo largo del tiempo de simulación especificado

Con esta información, al ejecutar el simulador se obtiene el comportamiento que se espera que tuviese el sistema simulado con las entradas especificadas.

Existen lenguajes de simulación para varios tipos de sistemas, como por ejemplo, redes de ordenadores, circuitos electrónicos, etc.

Los lenguajes de simulación se utilizan para hacer pruebas de un sistema antes de su

implementación real, con el objetivo de depurar errores en el diseño del sistema y ahorrar costes. Este ahorro de costes viene motivado por dos motivos:

- Detectar errores con el sistema ya implementado obliga a volver a rediseñar y volver a implementar el sistema lo cual puede requerir descartar el sistema ya diseñado (o partes de él), con el consiguiente coste en medios materiales.

- Modificar el modelo del sistema descrito con el lenguaje de simulación es mucho más rápido (consiste simplemente en modificar el fichero de texto que contiene la descripción del sistema en el lenguaje de simulación en cuestión) que modificar la implementación del sistema (por ejemplo, si se trata del diseño de un chip, generar una nueva oblea de silicio con el chip para posteriormente probarlo), con lo que se ahorra tiempo en la detección de errores.

En esta práctica vamos realizar un compilador que dado un programa en el lenguaje fuente que se presenta a continuación, genera un programa en Java que al ejecutarlo realiza la simulación. Por lo tanto, el código Java que se genere es en si mismo un simulador para el programa en lenguaje fuente.

Descripción general del lenguaje de programación fuente (VHDL)

Empecemos con un pequeño programa de ejemplo:

```
entity ejemplo is
       port (X, Y:in bit; Z1, Z2: out bit);
end;

architecture struct of ejemplo is
begin
    Z1<= transport X and Y after 10 ns;
    Z2<= transport X or Y after 10 ns;
end;
```

Un programa en el lenguaje VHDL consta de 2 partes: `entity` y `architecture`. La `entity` define un nombre para el circuito digital a simular (`and_gate`), y declara cuáles son las señales de entrada (`X` e `Y`) y salida (`Z1` y `Z2`). Toda señal tiene un tipo de datos asociado, si bien en la práctica sólo se considera un tipo de datos: el tipo `bit`. Por lo tanto, no será necesario realizar chequeo de tipos.

La `arquitecture` describe el comportamiento del circuito digital. En VHDL se permite definir varias `arquitecture` equivalentes para una misma `entity` (por ejemplo, varias descripciones del mismo sistema a diferentes niveles de abstracción). Por este motivo, se le da un nombre a la `architecture` (`struct`) y se indica a que `entity` corresponde una cierta `architecture` por medio del identificador que da nombre a la `entity` (`and_gate`). Nosotros no haremos uso de esta posibilidad de VHDL, sino que siempre definiremos una única `architecture` para cada `entity`.

Dentro de una `architecture` se pueden definir nuevas señales internas al circuito digital que se está definiendo (esta posibilidad no se utiliza en este ejemplo, pero si se debe soportar en la práctica).

El comportamiento del circuito se define por medio de una secuencia de sentencias de asignación de señales en el cuerpo de la `architecture` (entre el `begin` y el `end`).

Por ejemplo, la primera sentencia de asignación de señales del ejemplo es:

```
Z1<= transport X and Y after 10 ns;
```

Cada sentencia de asignación de señales tiene la siguiente forma:

1. Un identificador (en el ejemplo anterior, `Z1`). Una sentencia de asignación de señales define el valor que toma una determinada señal a lo largo del tiempo. Esta señal es aquella cuyo nombre es el identificador al comienzo de la sentencia.

2. La cadena "`<= transport`". La palabra clave `transport` hace referencia al algoritmo utilizado para calcular la evolución de la señal. Hay dos tipos de sentencias de asignación de señales en VHDL: `transport` e `intertial`. En la práctica sólo se consideran sentencias tipo `transport`.

3. Una expresión que define el valor que toma la señal asignada (en el ejemplo `X and Y`).

4. La cadena "`after`" indica que a continuación se indica el retardo que tarda el valor calculado con la expresión anterior en propagarse a la señal asignada.

5. Una expresión temporal que define el retardo (en el ejemplo `10 ns`). En la práctica sólo se consideran expresiones temporales formadas por una constante entera estrictamente mayor que 0, seguidas de la cadena "`ns`" (nanosegundos).

Definición de la sintaxis del lenguaje de programación fuente

A continuación vamos a dar la definición del lenguaje fuente en BNF. Previamente definimos el alfabeto del lenguaje:

```
A={a, b, ..., z, A, B, ..., Z, 0, 1, ..., 9,
";", "<", "=", "(", ")", ":", ",", "_", "'"}
```

El lenguaje no distingue mayúsculas y minúsculas; por lo tanto, los identificadores `Hola` y `hola` son iguales. Las palabras clave también se pueden escribir utilizando mayúsculas y minúsculas indistintamente. Las comillas ("") se utilizan para rodear símbolos no alfanuméricos.

La definición del lenguaje pedido en BNF es la siguiente:

```
<Design_unit> ::= <Entity_declaration> <Architecture_body>

<Entity_declaration> ::= entity <Ident> is <Port_clause> end;

<Port_clause> ::= port (<Interface_list>);

<Interface_list> ::= <Interface_element>
                | <Interface_element>; <Interface_list>

<Interface_element> ::= <Identifier_list>: <Mode> bit

<Mode> ::= in | out

<Identifier_list> ::= <Ident>
                 | <Ident>, <Identifier_list>

<Architecture_body> ::= architecture <Ident> of <Ident> is
                   [<Signal_declaration>]
                   begin <Architecture_statement_part> end;
```

```
<Signal_declaration> ::= signal <Identifier_list>: bit;

<Architecture_statement_part> ::= <Concurrent_statement>
                  | <Concurrent_statement> <Architecture_statement_part>

<Concurrent_statement> ::= <Ident> <= transport <Expr> after <CEnt> ns;

<Expr> ::= <Expr> and <Expr>
       | <Expr> or <Expr>
       | <Expr> xor <Expr>
       | <Expr> nor <Expr>
       | <Expr> nand <Expr>
       | (<Expr>)
       | not <Expr>
       | <Ident>
       | <Clog>

<CLog> ::= '1' | '0'

<Digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Letra> ::= a | b | ... | z | A | B | ... | Z

<CEnt> ::= <Digito>
          | <Digito> <CEnt>

<Ident> ::= <Letra>
           | <Letra> <ListaLetraODigito>

<LetraODigito> ::= <Letra>
                  | <Digito>
                  | _

<ListaLetraODigito> ::= <LetraODigito>
                  | <LetraODigito> <ListaLetraODigito>
```

Expresiones en el lenguaje de programación fuente

El subconjunto del lenguaje VHDL considerado incluye los operadores lógicos `and`, `or`, `not`, `xor`, `nand` y `nor`. Los operadores `and`, `or`, `xor`, `nand` y `nor` tienen la misma precedencia, que es menor que la del operador `not`. Los operadores `and`, `or` y `xor` asocian por la izquierda. No se permite una expresión con una secuencia (sin paréntesis) de operadores, a menos que todos los operadores sean iguales (se permite además el uso de operadores `not`) y uno de `and`, `or` o `xor`. Por ejemplo, son expresiones lógicas válidas las siguientes:

- `a and b and c`
- `a and not b and c`
- `not a and not b and not c`
- `a or b or c`
- `a xor b xor c`
- `(a and not b) or c`
- `(a nand b) nand c`

- `a nand (b nand c)`
- `(not a nand b) nor c`

Son expresiones lógicas **incorrectas** las siguientes:

- `a and not b or c`
- `a nand b nand c`
- `not a nand b nor c`

La identificación de qué expresiones lógicas son correctas o no de acuerdo con lo expuesto en este apartado **puede y debe hacerse** en la fase de análisis sintáctico. Nótese que la definición BNF que se ha dado no cumple esta condición.

Requisitos que deben cumplir los ficheros JLex y CUP

Para la prueba del analizador léxico y el analizador sintáctico, se utilizara la clase Main que se proporciona. La ejecución de dicha clase se realizará de acuerdo con lo siguiente:

```
java Main <nombre_fichero>
```

Donde `<nombre_fichero>` es el nombre del fichero (con el path si es necesario) del programa fuente a analizar.

El programa deberá de levantar una excepción que no deberá de ser capturada en el caso de que el programa fuente tenga algún error de sintaxis. Los mensajes emitidos por excepciones producidas por errores en la fase de análisis sintáctico deberán indicar una línea del programa orientativa del lugar en el que se ha producido el error.

Obligatoriamente las clases generadas por CUP deberán pertenecer a un paquete llamado Parser y la clase generada por JLex deberá pertenecer a un paquete llamado Lexer.

Ayudas y sugerencias

Se proporciona un juego de tests con el que probar la práctica. De entre ellos solamente deberían de producir un error de sintaxis los tests de `Error1` a `Error8`, ambos inclusive (los tests de `Error9` a `Error14` tienen errores semánticos pero no errores sintácticos).

Dado que no es necesario que el analizador sintáctico produzca el árbol de sintaxis abstracta y no se entregan clases Java no deberá asociar clases Java a ningún símbolo no terminal y no deberá incluir construcciones `{: ... :}` en el fichero CUP. Sin embargo, puede, si lo desea, asociar clases Java a los terminales.

Recuerde que debe de compilar las clases del analizador sintáctico antes de compilar la clase del analizador léxico, ya que esta importa la clase `sym` generada por CUP.

Ficheros a entregar

Se deberán entregar exclusivamente los siguientes ficheros:

- fichero `Yylex` en formato JLex para el analizador léxico

- fichero `parser` en formato CUP para el analizador sintáctico

Criterios de corrección

Esta práctica tiene un peso de 2.5 puntos. Como regla general se restarán 0.5 puntos por cada test fallado. Si se fallan más de 3 tests se perderá automáticamente la convocatoria de Junio.

Localización | Personal | Docencia | Investigación | Novedades | Intranet
inicio | mapa del web | contacta

Last Revision: 04/03/2004 10:44:44