Análisis Sintáctico

Preguntas más frecuentes

SIN.01. ¿Por dónde empiezo?

R. Si usted ha llegado hasta aquí es porque ya ha construido el analizador léxico y éste es capaz de generar secuencias ordenadas de tokens. Ahora es momento de construir el analizador sintáctico para que vaya solicitando tokens al analizador léxico bajo demanda según va recorriendo las reglas de producción gramaticales. Para conseguir el adecuado nivel de integración entre el analizador léxico y el sintáctico, es preciso seguir una serie de pasos y convenciones que se explican en la pregunta SIN.02. Pero antes debería preocuparse de escribir la gramática en la especificación de Cup. Para ello, recomendamos que siga los siguientes pasos (si tiene dudas de cómo llevar a cabo alguno de ellos consulte el manual de Cup):

1. Estudie las especificaciones del lenguaje y defina como terminales todos los tipos de Tokens que genere su analizador léxico. Tipifique esos token como pertenecientes a la clase Token (por ejemplo: terminal Token MAS).

2. Estudie las especificaciones del lenguaje y, en función del tipo de práctica (A o B), escriba un ejemplo de código fuente en papel lo suficientemente complejo como para que contemple todas las construcciones sintácticas de la especificación.

3. Realice una descomposición modular de ese código fuente en partes (ayúdese de marcadores de colores). El análisis debería llevarle a conclusiones del estilo: “Un programa en pascal está compuesto de una sección de declaración, una sección de subprogramas y un programa principal”; “Una sección de declaraciones está compuesta de una sección opcional para la declaración de constantes, otra para la de tipos y otra para la de variables”, “Una sección de declaración de constantes está formada por la palabra reservada CONST seguida de una colección de de declaración de una constante separadas por punto y coma”; “una declaración de constante está formada por un identificador seguido de un igual y de un valor numérico entero o lógico (TRUE o FALSE)”…

4. Asigne un nombre significativo a cada una de las partes identificadas en la descomposición anterior. Por ejemplo, declaracionConstantes, declaraciónContante, ListaIdentificadores, Tipo.

5. Traduzca esa descomposición a reglas de producción e insértelas en el fichero de especificación de Cup. Por ejemplo:

```
seccionConstantes ::= CONST declaracionConstantes;
declaracionConstantes ::= declaracionConstante |
                          declaracionConstante declaracionConstantes;
declaracionConstante ::= listaConstantes IGUAL valorConstante;
listaConstantes ::= …
```

6. Proceda así recursivamente hasta construir todas las reglas de producción gramatical

7. Revise la gramática para comprobar que no hay redundancias en la declaración. Por ejemplo, en el caso anterior descubrirá que listaConstantes tiene una definición gramatical idéntica a las listas de parámetros utilizadas en las declaraciones de subprogramas y también igual a la de lista de variables de la declaración de variables. Por tanto conviene sustituir las reglas para listaConstantes, listaParametros y listaVariables por listaIdentificadores, no terminal más abstracto que los engloba a todos. Existen otros ejemplos de refactorización típicos que se pretende que el alumno descubra.

8. Corrija el diseño gramatical para eliminar todo tipo de conflictos. Consulte las preguntas SIN.19 a SIN.27.

SIN.02. ¿Cómo se integran JFlex y Cup?

R. Una vez obtenida la especificación Cup, se debe ejecutar la tarea ant para que genere el parser. Si ésta no contiene errores se debería obtener una clase parser.java y otra Sym.java. Esta última contiene constantes publicas java para cada uno de los terminales declarados en la especificación de Cup (recuerde que los terminales del análisis sintáctico son los tokens del análisis léxico). Ahora vuelva a la especificación JFlex y asegúrese de que el objeto Token que se construye dentro de cada regla patrón - acción de JFlex incluye como identificador el valor de la constante adecuada de la clase Sym.

```
"+"      { Token token = new Token (sym.PLUS);
           token.setLine (yyline + 1);
           token.setColumn (yycolumn + 1);
           token.setLexema (yytext ());
           return token;
         }
```

SIN.03. ¿Hay que hacer las tablas del análisis sintáctico?

R. No. El trabajo consiste en especificar en la herramienta Cup la gramática de acuerdo a las especificaciones del lenguaje y dejar que ella construya un analizador sintáctico automáticamente.

SIN.04. ¿Qué son expresiones?

R. Una expresión es una construcción sintáctica del lenguaje que devuelve al contexto donde se evalúa un valor. Son expresiones los valores constantes de cualquier tipo, los identificadores de constantes, variables enumeraciones y parámetros, los elementos de vector, los campos de registros, y las llamadas a funciones. También son expresiones cualquier combinación sintácticamente correcta de expresiones utilizando operadores aritméticos, lógicos, relacionales y expresiones entre paréntesis. (Esta definición puede depender del lenguaje en concreto).

SIN.05. ¿Qué tipos de expresiones se distinguen?

R. Por su tipo distinguimos dos tipos de expresiones: 1) expresiones aritméticas, que devuelven un valor numérico al evaluarse y 2) expresiones lógicas, que devuelven un valor de verdad. Son expresiones aritméticas los valores constantes, identificadores (de constantes, variables y parámetros de tipo entero o real), llamadas a funciones que devuelven un tipo entero o real y cualquier combinación de expresiones aritméticas con operadores aritméticos (suma resta, producto, cociente, resto…). Son expresiones lógicas los valores TRUE y FALSE, los identificadores (de constantes, variables y parámetros de tipo lógico), llamadas a funciones que devuelven un tipo lógico, cualquier combinación de expresiones aritméticas con operadores lógicos (y, o, no, etc.) y la combinación de expresiones aritméticas mediante operadores relacionales (igual, distinto, mayor, menor, etc.). (Esta definición puede depender del lenguaje en concreto).

SIN.06. ¿Es conveniente distinguir gramaticalmente entre expresiones aritméticas y expresiones lógicas?

R. En principio parece pensar que es una buena idea. Dada la siguiente gramática:

exp ::= expLogica | expArit | ID

expLogica ::= TRUE | FALSE | expLogica AND expLogica | expLogica OR expLogica

expArit ::= expArit MAS expArit | expArit POR expArit |ID

Al encontrar la frase “a AND b” el parser sabrá reconocerla como expresión lógica debido a la existencia del operador AND. Por el mismo motivo “a * b” será identificado como una expresión aritmética por la aparición del operador POR (*). Pero, ¿y si la frase es simplemente “a” (como por ejemplo al procesar la parte derecha de la asignación “b:=a”)? Entonces la gramática no tiene información contextual suficiente para determinar el tipo de expresión (aritmética o lógica que representa “a”). Por eso este tipo de disquisiciones es frecuente tratarlas en la siente fase de análisis semántico.

SIN.07. ¿Cómo se inyecta la información de asociatividad y precedencia en la gramática?

R. En gramáticas LL(1), utilizadas en los analizadores descendentes, la información de asociatividad y precedencia se inyecta en la propia gramática utilizando no terminales (típicamente E, T y F) para distinguir diferentes niveles de asociatividad. Esto genera gramáticas complejas aunque no ambiguas. En Cup se recomienda usar la gramática de operadores clásica que es inherentemente ambigua:

```
exp ::= exp MAS exp |
        exp POR exp |
        LBRACKET exp RBRAKET |
        ID | NUMERO;
```

Para resolver la ambigüedad existe una sección en Cup donde se indica explícitamente la información de asociatividad y precedencia de los operadores (MAS, POR, LBRACHET…). Esta información es utilizada por Cup para determinar que regla de derivación utilizar ante un punto de indecisión motivado por el carácter ambiguo de la gramática

SIN.08. ¿Los parámetros por referencias pueden ser expresiones?

R. Debido a que los parámetros por referencia son de salida o entrada / salida estos deben ser identificadores. Por tanto, no pueden ser expresiones.

SIN. 09. ¿Qué es la precedencia de operadores y que implica?

R. La precedencia de operadores indica el orden en que se deben evaluar las expresiones. Gracias a ella por ejemplo podemos afirmar que la construcción “2+3*5” debe calcularse siempre evaluando primero el 3*5 y al resultado sumarle el 2 y nunca hacer 2+3 y multiplicar después el resultado por 5. El orden en el que un analizador sintáctico sin información de precedencia y con una gramática ambigua como la siguiente realiza las operaciones no está determinado.

```
R1. exp ::= exp MAS exp
R2. exp ::= exp POR exp
R3. exp ::= NUMERO;
```

Esto es debido a que dicho orden viene condicionado por la forma en que se construye el árbol de derivación que a su vez depende de la selección particular de reglas que haga el parser en dicho proceso. Por ejemplo si primero aplicamos la regla de derivación R1 para cubrir la suma y luego la regla R2 Para cubrir el producto y finalmente aplicamos varias veces la regla R3 para obtener los números 2, 3, y 5 obtendremos un árbol semánticamente correcto de acuerdo a la interpretación matemática de precedencia de los operadores aritméticos. No obstante, la misma frase también podría haberse parseado escogiendo primero R2 para cubrir el producto y luego R1. Aunque ambas opciones dan lugar a sendos árboles de derivación sintácticamente correctos, cada uno de ellos imprime un orden de aplicaciones de los operadores diferente (el producto antes que la suma en el primer caso y al revés en el segundo). Por tanto aunque el segundo árbol de derivación es sintácticamente correcto resulta semánticamente inválido de

acuerdo a los convenios matemáticos relativos a la precedencia del producto con respecto a la suma.

SIN 10. ¿Qué significa que existan operadores de igual nivel de precedencia (suma y resta o producto y división?

R. Significa que el orden en que se realicen las operaciones es irrelevante. Por ejemplo para "3+2-1" es irrelevante hacer 3+2 y al resultado restarle 1 o hacer 2-1 y al resultado sumarle 3. (Un caso similar puede encontrarse para productos y cocientes por ejemplo). Debido a ello no es necesario imponer ningún orden de precedencia en la expresión gramatical entre estos pares de operadores.

SIN.11. ¿Qué significa que los paréntesis tengan información de precedencia?

R. La parentización se utiliza para alterar el orden de evaluación de las expresiones y modificar así el resultado. Por ejemplo, si 2+3*5 es igual a 17 ya que la precedencia del producto frente a la suma impone computar primero 3*5 y después sumar al resultado 2, al incluir los paréntesis de la forma (2+3)*5 el resultado obtenido será 25 ya que se altera el orden de aplicación de los operadores haciendo en este caso más prioritario la evaluación de la suma frente al producto. Para conseguir este efecto de parentización en un compilador debemos imponer que los paréntesis sean los operadores (de parentización) más prioritarios. Es decir, los paréntesis (tanto izquierdo como derecho) siempre tienen un orden de precedencia mayor que cualquier otro operador.

SIN.12. ¿Que son las sentencias?

R. Las sentencias son construcciones sintácticas del lenguaje con una semántica operacional determinada y que no devuelven ningún valor al contexto de evaluación. Son ejemplos de sentencias las de control de flujo condicional (if, case,…), las de control de flujo iterativo (for, while,…), las llamadas a procedimientos y las sentencias de asignación. (Esta definición puede depender del lenguaje en concreto).

SIN.13. En lenguajes de tipo C, ¿Debe tratarse la función main de forma especial o puede ser considerada como otra función más?

R. Existen dos opciones igualmente válidas:

1. Definir la palabra reservada main para forzar la existencia de la misma una y solo una vez en el programa e incluir reglas gramaticales específicas para ella (téngase en cuenta que el prototipo de la función mail es uno específico). Esta opción diferencia el tratamiento mail del resto de funciones
2. Tratarlo como una función más. Para ello main no debe estar entre el conjunto de palabras reservadas del lenguaje (sería un identificador de función más. En este caso, se posterga para la fase de análisis semántico la existencia y unicidad de la función main.

SIN.14. La sentencia RETURN ¿puede ir en cualquier lugar de las instrucciones de una función o ha de ser la última instrucción?

R. Sí.

SIN.15. ¿Una función puede tener varias sentencias RETURN?

R. Sí.

SIN.16. ¿Cómo puedo comprobar que cada función deba tener al menos una sentencia RETURN alcanzable para cada posible ejecución de la misma?
R. Difícilmente. Recomendamos que este tratamiento se postergue a la fase de análisis semántico.

SIN.17. En lenguajes de tipo MODULA, ¿Cómo se comprueba que el id de module coincide con el del end del module?
R. Difícilmente. Recomendamos que este tratamiento se postergue a la fase de análisis semántico.

SIN.18. ¿En el análisis sintáctico se ha de comprobar que las variables y llamadas a funciones del cuerpo del programa han sido previamente declaradas y pertenecen a un ámbito permitido?
R. No. Eso es misión de la fase de análisis semántico.

SIN.19. ¿Cómo debo implementar la recuperación de errores sintácticos en Cup?

R. Para realizar la recuperación de errores en Cup es necesario ampliar la gramática incorporando nuevas reglas que contemplan las situaciones de error. Para ello se utiliza el símbolo no terminal especial 'error'. Por ejemplo, para una gramática de operadores:

```
exp ::= exp MAS exp |
        exp POR exp;
```

Deberemos añadir una nueva regla:

```
exp ::= exp MAS exp |
        exp POR exp |
        error;
```

Esta modificación se puede interpretar de la siguiente forma. 1) Si la secuencia de tokens a la entrada encaja con exp MAS exp, se procede por la primera regla y se realiza una reducción a exp. 2) Si por el contrario encaja con exp POR exp la reducción se lleva a cabo por la segunda regla. 3) En caso de que los tokens a la entrada no encajen con ninguna de las reglas anteriores Cup consumirá errores en estado de pánico hasta llegar a una nueva situación estable. Asimilando todos los tokens erróneos al no terminar el proceso de compilación continua ya que el error se reduce a expresión (aunque errónea) lo cual permite recuperarse de la situación de error.

SIN.20. ¿Existe algún modo en Cup de dar información adicional sobre errores sintácticos sin añadir reglas de error?

R. No.

SIN.21. ¿Cuándo se alcanza una nueva situación estable tras un error?

R. Una vez entrado en una producción de error, la consumición de tokens en estado de pánico continua hasta que se consuma un máximo de tokens o se encuentre un token de sincronización a la entrada (lo que ocurra primero). Los tokens de sincronización suelen ser delimitadores que marcan el final del error dentro del contexto. Por ejemplo:

```
exp ::= exp MAS exp |
        exp POR exp |
        error PYC;
```

Indica que en caso de error se consumirán tokens hasta que se alcance el siguiente token punto y coma ';' (Debe suponerse que en el ejemplo PYC representa el token punto y coma) .

SIN.22. ¿Es posible indicar varios tokens de sincronización para una producción de error?

R. Sí, simplemente deben colocarse tantas reglas de error como tokens de sincronización se deseen declarar. Por ejemplo, para una gramática tipo PASCAL podemos incluir el punto y coma, el cierre de paréntesis y la palabra clave END como 3 tokens de sincronización posibles.

```
exp ::= exp MAS exp |
        exp POR exp |
        error PYC |
        error RBRACKET |
        error END;
```

SIN.23. ¿Cuántos tokens se consumen como máximo en estado de pánico?

R. El número máximo que Cup consume como máximo en estado de pánico (al alcanzar un error) es 3. Este es un valor que puede configurarse. Consulte las transparencias de Cup para obtener detalles acerca de los parámetros de configuración de Cup.

SIN.24. ¿El orden en el que se insertan las producciones de error es irrelevante?

R. No. Cup va recorriendo las reglas de un mismo antecedente en el orden en que son definidas buscando un posible encaje con los tokens a la entrada. Por tanto, las reglas de error deben ser ubicadas siempre las últimas, como se ve en los ejemplos de este documento.

SIN.25. ¿Cómo puedo implementar el tratamiento de errores?

R. Dentro de la arquitectura proporcionada en la asignatura usted no tiene que hacer nada para implementar el tratamiento de errores. Esto es debido a que la plantilla cup proporcionada incorpora el siguiente código dentro de la sección parser code:

```
parser code {:
        SyntaxErrorManager syntaxErrorManager = new SyntaxErrorManager ();
        public void syntax_error (Symbol symbol) {
           Token token = (Token) symbol.value;
           syntaxErrorManager.syntaxError ("Error sintáctico", token);
        }
        public void unrecovered_syntax_error (java_cup.runtime.Symbol symbol) {
           Token token = (Token) symbol.value;
           syntaxErrorManager.syntaxFatalError ("Error fatal", token);
        }
:}
```

Como se puede observar este código implementa los manejadores de error (recuperable e irrecuperable) que son invocados automáticamente por la herramienta Cup cuando se alcanza una situación de error. La implementación que se hace consiste en extraer el token del contexto sintáctico que provoco el error y emitir un mensaje de error estándar. Se recomienda

no modificar esta implementación ya que la evaluación automática de la práctica se apoya en estas salidas.

SIN.26. ¿Cómo enriquecer el tratamiento de errores?

R. Si desea ampliar la información de error de forma genérica usted puede, sin alterar el código proporcionado, ampliarlo con nuevas llamadas al método info o debug del SyntaxErrorManager. Por ejemplo:

```
public void syntax_error (Symbol symbol) {
        Token token = (Token) symbol.value;
        syntaxErrorManager.syntaxError ("Error sintáctico", token);
        syntaxErrorManager.syntaxDebug ("Esto es un error…");
}
```

Si desea hacer un tratamiento de error específico al contexto donde este se produjo deberá enriquecer la gramática con acciones semánticas a tal efecto. Siempre que tenga que emitir un error utilice el SyntaxErrorManager y no la salida estándar de java:

```
exp ::= exp MAS exp |
        exp POR exp |
        error PYC {: syntaxErrorManager.syntaxDebug ("Esto es un error en exp…"); :};
```

SIN. 27. ¿Cómo puedo obtener el token que me ha causado un error?

R. Dentro de las reglas de producción de error no es posible obtener el token que causó un error. Este token se envía de forma automática a los manejadores de error por parte de Cup. Dentro de ellos sí es posible tratar el error con información de dicho token. Consulte la pregunta SIN.25.

SIN.28. ¿Es lo mismo un conflicto que un error sintáctico?

R. No. Los errores sintácticos son errores que un analizador sintáctico detecta al no coincidir los tokens a la entrada con ninguna parte derecha de ninguna regla de producción gramatical. Estos se producen durante el tiempo de compilación. Por el contrario los conflictos se producen en tiempo de diseño de un compilador debido a que en algún momento el compilador, para una secuencia de tokens a la entrada, no puede decidir inequívocamente porque regla de producción continuar el proceso de reducción – desplazamiento (derivación a la inversa).

SIN.29. ¿Qué tipos de conflictos existen?

R. Existen dos tipos de conflictos: 1) conflictos de reducción – reducción y 2) conflictos de reducción desplazamiento. Los primeros ocurren cuando la secuencia de token a la entrada encaja con la parte derecha de 2 o más reglas de producción imposibilitando al parser decidir qué reducción debe aplicar. Los segundos ocurren cuando la secuencia de tokens a la entrada puede ser reducida por una regla de producción pero también se ajusta a la parte derecha de otra regla más larga, lo que implicaría consumir más tokens hasta poder aplicar la reducción.

Ejemplo de conflicto reducción – reducción:

```
A ::= id; <-- Reduzco id a A ó
B ::= id; <-- Reduzco id a B
```

Ejemplo de conflicto reducción – desplazamiento.

```
A ::= id B;  <-- Desplazo para procesar B y reducir por A o
C ::= id;    <-- reduzco a C directamente
```

SIN.30. ¿Cómo se solucionan los conflictos?

R. En general no existen algoritmos para eliminar los conflictos que se producen cuando se diseña una gramática. La eliminación de de conflictos es una tarea de ajuste que requiere estudiar cada caso de conflicto en detalle y transformar la gramática a otra gramática equivalente (que reconozca el mismo lenguaje) para evitar el conflicto. Sin embargo si pueden darse algunas heurísticas de situaciones gramaticales arquetípicas que son fuentes potenciales de conflictos. En este sentido evite a toda costa gramáticas con fuentes de ambigüedad potenciales (consulte las transparencias), reglas cuya parte derecha comience por la misma secuencia ordenada de terminales (consulte factorización por la izquierda en las transparencias) o gramáticas reductoras (reglas con la palabra vacía épsilon). Por ejemplo:

```
L ::= id RESTO | id;
RESTO ::= COMA id RESTO | ;  <--- problema
```

Puede ser sustituida, equivalentemente, por las producciones siguientes:

```
L ::= id | id COMA L;
```

SIN.31. ¿Qué significan los warnings que genera Cup al construir el parser?

R. Con el fin de aligerar el proceso de diseño de analizadores sintácticos, muchas veces, cuando Cup encuentra conflictos de reducción - desplazamiento en lugar de generar un error asume una de las alternativas (generalmente siempre suele ser más conveniente desplazar antes que reducir en este tipo de conflictos.). En ese caso, emite un warning por la salida para informar de tal conflicto (y de tal asunción) al diseñador. En la medida de lo posible esos warnings deberían ser eliminados completamente sustituyendo las reglas de producción afectadas por otras equivalentes que eviten el conflicto. En cualquier caso siempre hay que asegurarse que la asunción hecha por Cup es la adecuada dentro del contexto gramatical en que se produce.

SIN.32. Cup genera muchos warnings al generar el analizar sintáctico y aborta el proceso con el mensaje "More conflicts enconuntered than expected". ¿Cómo puedo solucionar este problema?

R. En la medida de lo posible debe evitar que cup genere warnings. Para ello recomendamos construir la gramática paulatinamente en un ciclo iterativo de ampliación – depuración. Consulte la pregunta SIN.30 acerca de cómo se solucionan los conflictos. De todas formas existe la posibilidad de indicar a Cup que ignore los warnings a la hora de generar el parser. Para ello, introduzca los siguientes parámetros en la tarea ant de cup:

```
<target name="cup">
        destdir="${sourceBase}"
        dump="all"
        dump_tables="true"
        dump_grammar="true"
        dump_states="true"
       interface="true" progress="true"/>
<target>
```

SIN.33. Cuando introduzco caracteres especiales en un código fuente de prueba el proceso de compilación aborta anormalmente.

R. Debe indicar a JFlex que el juego de caracteres que constituye los ficheros de código fuente es ASCII extendido. Para ello utilice la directiva %% full de JFlex.

SIN.34. ¿Cómo puedo distinguir entre mayúsculas o minúsculas en Cup?

R. De ningún modo. La formación de tokens a partir de una secuencia ordenada de caracteres es responsabilidad del analizador léxico. Si necesita indicar que los caracteres de un tipo de token están formados por mayúsculas (o minúsculas) debe especificarlo en el fichero JFlex.

SIN.35. ¿Existe alguna herramienta para comprobar la corrección de gramáticas y comprobar paso a paso cómo se realiza en análisis sintáctico y la construcción del árbol de derivación para diferentes frases de entrada?

R. Jaciie es una de ellas y puedes encontrarla en la sección de herramientas. En http://webdiis.unizar.es/~ezpeleta/COMPI/compiladoresI.htm puedes encontrar una herramienta similar.