metacomp

Un generador de procesadores de lenguaje basados en analizadores recursivos descendentes

Procesadores de Lenguaje
Ingeniería Informática
Universitat Jaume I

Febrero de 2011
Versión 3.0β5

1. Introducción

Este documento es una introducción a metacomp, un generador de procesadores de lenguaje. A partir de la especificación de un analizador léxico y un esquema de traducción, metacomp proporciona un programa Python que analiza cadenas pertenecientes al lenguaje especificado por el analizador léxico y la gramática RLL(1) que sirve de soporte al esquema de traducción. A la vez que efectúa el análisis, el analizador ejecuta las acciones semánticas que se indiquen en el esquema de traducción.

1.1. Problemas de ejemplo

Para presentar metacomp desarrollaremos (parcialmente) dos sencillos ejemplos: una calculadora y un traductor de expresiones aritméticas a un lenguaje de pila. El lenguaje de entrada para ambos ejemplos viene descrito por la siguiente gramática:

$$
\begin{aligned}
\langle \text{Entrada} \rangle &\rightarrow (\langle \text{Sentencia} \rangle \texttt{;})^+ \\
\langle \text{Sentencia} \rangle &\rightarrow \langle \text{Expresion} \rangle (\texttt{->}\textbf{id})? \\
\langle \text{Expresion} \rangle &\rightarrow \langle \text{Termino} \rangle (\textbf{op\_sum} \langle \text{Termino} \rangle)^* \\
\langle \text{Termino} \rangle &\rightarrow \langle \text{Factor} \rangle (\textbf{op\_mult} \langle \text{Factor} \rangle)^* \\
\langle \text{Factor} \rangle &\rightarrow \textbf{ent} | \textbf{real} | \textbf{id} | \texttt{(} \langle \text{Expresion} \rangle \texttt{)}
\end{aligned}
$$

Las categorías léxicas que intervienen son:

Categoría	Expresión regular	Significado
ent	[0–9]$^+$	Números enteros
real	[0–9]$^+$\.[0–9]$^+$	Números reales
id	[a–zA–Z][a–zA–Z0–9]*	Identificadores
op_sum	[-+]	Operadores aditivos
op_mult	[*/]	Operadores multiplicativos

Además, tenemos las siguientes categorías "anónimas": "`->`", "`;`", "`(`" y "`)`". Estas categorías tienen asociadas un único lexema. Su expresión regular es trivial y las trataremos aparte en la especificación.

La calculadora (o *intérprete* de expresiones aritméticas) evalúa las expresiones aritméticas y muestra su resultado por pantalla cada vez que encuentra un punto y coma. Si la expresión va seguida de una flecha (`->`), el resultado se asigna (tras mostrarlo por pantalla) a la correspondiente variable. A partir de la primera vez que se asigna un valor a una variable, ésta puede participar en cualquier expresión. El tipo de la variable será el de la expresión que le dé valor (no lo comprobaremos en los ejemplos, puedes hacer como ejercicio que tu intérprete dé un error si hay violación de tipos). Si en una expresión sólo participan números enteros, el resultado es un número entero. La operación de un entero con un real (o de un real con un real) devuelve un número real.

El traductor de expresiones (o *compilador* de expresiones aritméticas) debe producir un programa ejecutable para el ensamblador de una sencilla calculadora de pila. Las siguientes instrucciones de la calculadora operan con valores enteros:

- `PUSH` *entero*: apila un número entero.
- `PUSH &`*dir*: apila el número entero almacenado en la dirección *dir*.
- `POP &dir`: desapila un número entero y lo guarda en la dirección *dir*.
- `PRINT`: muestra por pantalla un número entero.
- `ADD`: desapila dos enteros de la pila, los suma y apila el resultado.
- `SUB`: desapila dos enteros de la pila, resta el que estaba más arriba del que estaba más abajo y apila el resultado.
- `MUL`: desapila dos enteros de la pila, los multiplica y apila el resultado.
- `DIV`: desapila dos enteros de la pila, divide el de más abajo entre el que está más arriba y apila el resultado.

Todas estas instrucciones pueden empezar por la letra `R`, en cuyo caso operan con reales. Por otra parte, la instrucción `ITOR` convierte el número entero que ocupa la cima de la pila en un número real. Las reglas que seguiremos para los tipos serán las mismas que para el intérprete.

1.2. Organización del documento

La segunda sección presenta la estructura de las especificaciones que metacomp toma como entrada (que denominaremos "programas metacomp"). La tercera sección enseña el modo en que debe codificarse un analizador léxico. La notación con que especificamos gramáticas incontextuales se presenta en la cuarta sección. La quinta sección se dedica a presentar la sintaxis de las acciones semánticas. El tratamiento de errores se explica en la sección sexta.

La séptima sección se dedica a presentar algunas posibilidades de metacomp que pueden ser de utilidad y también algunas de las limitaciones que debemos tener en cuenta.

Finalmente, los apéndices presentan un borrador de manual de referencia de metacomp y una sección de preguntas y respuestas.

2. Formato de los programas metacomp

Un programa metacomp se divide en varias secciones. Cada sección se separa de la siguiente por un carácter "`%`" que ocupa la primera posición de una línea. La primera sección contiene la especificación del analizador léxico. Las secciones pares contienen código de usuario. La tercera y sucesivas secciones impares contienen el esquema de traducción. Es decir, un programa metacomp se escribe en un fichero de texto siguiendo este formato:

Especificación léxica
`%`
Código de usuario
`%`
Esquema de traducción
`%`
Código de usuario
`%`
Esquema de traducción
⋮

Las secciones de "código de usuario" se utilizan para declarar estructuras de datos y variables, definir funciones e importar módulos útiles para el procesador de lenguaje que estamos especificando. Estos elementos deberán codificarse en el lenguaje de programación Python. La herramienta metacomp copia literalmente estos fragmentos de código en el programa Python que produce como salida.

Por defecto, metacomp espera que el programa fuente esté codificado en UTF-8 y genera programas en esa misma codificación. Sin embargo, se pueden cambiar las dos codificaciones de manera independiente. La opción "`-C`" (o "`--codEntrada`") cambia la codificación del fichero de entrada, mientras que la opción "`-c`" (o "`--codSalida`) cambia la del fichero de salida.

3. Especificación del analizador léxico

La sección de especificación léxica de un programa metacomp consiste en una sucesión de líneas, cada una de las cuales está en blanco, comienza por el carácter "`#`" o define una categoría léxica. En los dos primeros casos, la línea no es tenida en cuenta por metacomp. La sección finaliza con una línea que empieza con el carácter "`%`" e indica el inicio de una sección de código de usuario.

Una definición de categoría léxica consta de tres elementos (separados por blancos): un identificador de categoría léxica, una función de tratamiento del lexema y una expresión regular. Describimos con más detalle cada uno de estos elementos:

- El identificador de categoría es un cadena de caracteres formada por letras minúsculas, mayúsculas y/o el carácter de subrayado (`_`). El identificador no puede tener el prefijo `mc_`, que está reservado para las categorías que genera metacomp (actualmente sólo `mc_EOF`). Tampoco puede terminar en el carácter subrayado ni coincidir con ninguna palabra reservada de Python. El identificador `None` es especial: indica que los lexemas pertenecientes a esta categoría juegan el papel de "blancos", esto es, el analizador léxico no deberá emitir componente léxico alguno al detectarlos.
- La función de tratamiento del lexema es el nombre de una función Python definida en alguna de las secciones de código de usuario y tiene por objeto procesar el lexema para extraer de él la información que se precise (por ejemplo, su valor numérico en el caso de lexemas clasificables como números enteros). Si no deseamos procesar el lexema cuando se detecta un componente léxico de esta categoría, lo indicaremos poniendo `None` en lugar de un nombre válido de función. La función recibe un sólo parámetro: el componente léxico que devolverá el analizador léxico. Los componentes léxicos tienen tres atributos predefinidos:
  - `cat`: la categoría del componente.
  - `nlinea`: el número de línea donde se ha encontrado el componente.
  - `lexema`: el lexema del componente.

  La función puede añadir los atributos que necesite o cambiar alguno de los anteriores (aunque probablemente sólo tenga sentido cambiar el atributo `cat`).
- Finalmente, la expresión regular debe seguir la sintaxis descrita en el apéndice A.2.

El analizador léxico generado por metacomp segmenta la entrada buscando repetidamente el prefijo más largo compatible con alguna de las expresiones regulares. Puede suceder que la entrada tenga un prefijo que sea aceptado por más de una de las expresiones regulares y que este prefijo sea el más largo de entre todos los aceptados por estas expresiones. La solución que da metacomp a este conflicto consiste en asignar al prefijo la categoría léxica que aparece primero en la especificación.

NUEVO (β3)

No hace falta especificar explícitamente aquellas categorías que tienen sólo un lexema posible; se pueden incorporar dentro de las reglas de la gramática. Más adelante se comenta algunas de sus peculiaridades.

Podemos codificar el analizador léxico de nuestros dos ejemplos del siguiente modo:

```
1  # Blancos:
2  None    None    [ \t\n]+
3
4  # Operadores:
5  op_sum  None    [+-]
6  op_mult None    [*/]
7
8  # Operandos:
9  id      None    [a-zA-Z][a-zA-Z0-9]*
10 entero  l2int   [0-9]+
11 real    l2real  [0-9]+\.[0-9]+
12
13 %
14
15 def l2int(componente):
16   componente.v = int(componente.lexema)
17
18 def l2real(componente):
19   componente.v = float(componente.lexema)
```

La primera categoría léxica que definimos corresponde a los espacios en blanco, que son descartados por el analizador léxico (el identificador de categoría utilizado es `None`). Los espacios en blanco no necesitan efectuar tratamiento alguno sobre el lexema, por lo que hemos indicado con `None` que no hay función de tratamiento.

Los operadores de suma y resta pertenecen a la categoría léxica `op_sum`. Para distinguir posteriormente qué operador desencadenó la detección de un componente léxico de categoría `op_sum`, utilizaremos el lexema. Dado que éste se almacena automáticamente en el componente, no necesitamos ninguna función de tratamiento. De manera similar tratamos los operadores de multiplicación y división. Los identificadores tampoco necesitan tratamiento especial ya que sólo estaremos interesados en sus lexemas. En cuanto a los números enteros y reales, puedes ver que lo que hacemos es crear el atributo valor (`v`) a partir del lexema del número. Para ello utilizamos las funciones `l2int` y `l2real` respectivamente. Si hubiéramos deseado incluir algún tratamiento de error (por ejemplo, comprobar que los rangos son correctos), lo habríamos añadido a estas funciones.

No hemos incluido la especificación de los terminales punto y coma, paréntesis abierto y cerrado y asignación. Luego veremos cómo se incluye su definición directamente en las líneas de la gramática.

Como la especificación léxica es válida tanto para el intérprete como para el compilador, la copiamos en sendos ficheros `interprete.mc` y `compilador.mc`. Usamos la terminación "`.mc`" para identificar los ficheros que contienen programas metacomp. Aún no podemos procesar estos ficheros con metacomp, pues falta incluir en ellos sus correspondientes esquemas de traducción.

4. Especificación de la gramática incontextual

El siguiente paso es especificar la gramática incontextual que constituye el soporte del esquema de traducción. La gramática es una sucesión de producciones *terminadas* por punto y coma. Cada producción consta de una parte izquierda y una parte derecha, separadas entre sí por el símbolo "`->`". La parte izquierda consiste en un único símbolo no terminal. La parte derecha es una expresión regular (posiblemente vacía) de símbolos terminales y no terminales. Los elementos que se pueden emplear para construir la expresión regular son:

Secuencias Se indican simplemente mediante la concatenación de sus componentes.

Disyunciones Se indican separando las distintas opciones mediante barras verticales (`|`).

Clausuras Se indican encerrando entre paréntesis el componente que se puede repetir y con un asterisco (`*`) o una cruz (`+`) tras los paréntesis. El asterisco permite cero o más repeticiones mientras que la cruz (clausura positiva) permite una o más repeticiones.

Partes opcionales Se indican encerrando el componente que puede aparecer o no y con un interrogante (`?`) tras los paréntesis.

Los símbolos terminales son o identificadores de categoría léxica definidos en la especificación léxica o categorías definidas directamente en las reglas. Para especificar una categoría directamente en las reglas, escribimos el único lexema de la categoría entre comillas dobles (`"`). Las categorías que se definen directamente en las reglas tienen algunas restricciones:

- Sólo tienen un posible lexema, que es el que se especifica al definir la categoría.
- No es posible referirse directamente a estos terminales en las acciones semánticas.
- La categoría de estos componentes es su lexema precedido por un símbolo de admiración (`!`).

Además, la cadena que los especifica no puede contener comillas dobles y las secuencias de escape que contengan no son interpretadas (una secuencia como `\t` se interpretaría como una barra seguida de una te). En caso de conflicto entre una categoría definida en la especificación y una definida directamente, se da prioridad a la definida directamente.

Los símbolos no terminales son cadenas de caracteres formadas por letras minúsculas, letras mayúsculas y/o el carácter de subrayado no terminadas en subrayado y encerradas entre los caracteres "`<`" y "`>`".

Se pueden incluir comentarios en la gramática; éstos comienzan con el carácter "`#`" y terminan en el final de la línea correspondiente.

Podemos escribir la gramática presentada en la primera sección así:

```
<Entrada> -> ( <Sentencia> ";" )+ ;
<Sentencia> -> <Expresion> ( "->" id )? ;
<Expresion> -> <Termino> ( op_sum <Termino> )* ;
<Termino> -> <Factor> ( op_mult <Factor> )* ;
<Factor> -> ( entero | real | id | "(" <Expresion> ")" )
```

Si hubiéramos necesitado codificar una regla con la parte derecha vacía ($\langle A \rangle \rightarrow \lambda$) habríamos utilizado una parte derecha vacía (`<A> -> ;`).

Dado que esta gramática sirve de soporte tanto para el esquema de traducción del intérprete como para el del compilador, la añadimos (tras una línea cuyo primer carácter es "`%`") a los ficheros `interprete.mc` y `compilador.mc`.

Podemos generar ya un analizador sintáctico (con su correspondiente analizador léxico) para el intérprete ejecutando la orden

```
metacomp interprete.mc
```

Por pantalla veremos aparecer el código Python generado por metacomp. Si deseamos almacenar la salida en un fichero, podemos utilizar la opción "`-s`" (o "`--salida`") seguida de un nombre de fichero:

```
metacomp interprete.mc -s interprete.py
```

5. Especificación de las acciones semánticas

El código generado por metacomp se limitará a analizar cadenas válidas para nuestra gramática, pero no hará nada más a menos que añadamos acciones semánticas a la gramática incontextual. Una acción semántica en un programa metacomp es un fragmento de código Python encerrado entre dos arrobas (carácter "`@`") y sin caracteres fin de línea entre ellas. Las acciones semánticas pueden aparecer en cualquier posición de la parte derecha de una producción.

Si sustituimos la primera producción de la gramática por esta:

```
<Entrada> -> (
               <Sentencia> ";"
               @print "Expresión completa."@
             )+
             @print "Final de fichero."@
             ;
```

nuestro analizador debería, en principio, mostrar el mensaje "`Expresión completa.`" cada vez que encuentre un punto y coma; imprimirá, además, el texto "`Final de fichero.`" al encontrar el final del fichero. Podemos probarlo compilando `interprete.mc` sobre `interprete.py` y ejecutando éste último. Si creamos un fichero `expresiones` con las siguientes líneas:

```
2+2 -> i;
3*i;
```

y ejecutamos `interprete.py < expresiones` obtenemos la siguiente salida por pantalla:

```
Expresión completa.
Expresión completa.
Final de fichero.
```

Los ficheros generados por metacomp pueden utilizarse de dos formas distintas. Una posibilidad es usarlos como módulos de los que se pueden importar las clases `AnalizadorSintactico` y `AnalizadorLexico` (no es aconsejable hacer un "`import *`" de un fichero generado por metacomp). La otra posibilidad es utilizarlos directamente como ficheros ejecutables. En este último caso, el programa generado analiza su entrada estándar. Para cambiar este comportamiento, hay que definir la función `main`. Así, si queremos que nuestro intérprete lea de la entrada estándar si no tiene parámetros o de los ficheros que se le pasan como parámetros, podemos incluir al final de `interprete.mc` el siguiente código:

```
92 %
93
94 def main():
95   if len(sys.argv) ==  1:
96     AnalizadorSintactico(sys.stdin)
97   else:
98     for f in sys.argv[1:]:
99       AnalizadorSintactico(open(f))
```

5.1. Atributos

En las acciones semánticas contamos con la posibilidad de crear y utilizar atributos asociados a los símbolos que aparecen en una producción y que no corresponden a categorías anónimas. Por ejemplo, para la producción

```
<A> -> <B> c <D> e ;
```

contamos con una serie de variables predefinidas: `A`, `B`, `c`, `D` y `e`, a cada una de las cuales podemos asociar cuantos atributos queramos. Un atributo `atr` asociado a `A` se denota con `A.atr`. Las variables que corresponden a terminales llevan predefinidos los atributos creados por el analizador léxico y por la correspondiente función de tratamiento del lexema. Por ejemplo, en la producción de la gramática que nos sirve de ejemplo

```
<Factor> -> ent | real | id | "(" <Expresion> ")" ;
```

la variable `ent` tiene los atributos `v`, creado por la función `l2int`; y `cat`, `lexema` y `nlinea`, creados automáticamente por el analizador léxico.

Una advertencia es necesaria aquí. Las variables asociadas a los no terminales existen durante toda la ejecución de la regla. Esto es, podemos utilizarlas en acciones semánticas colocadas en cualquier posición de la regla. Sin embargo, las variables asociadas a los terminales existen únicamente a partir del momento en que se ha reconocido en la entrada el correspondiente terminal. Así, en la producción anterior, no es posible referirse a la variable `ent` más que en las acciones situadas tras `ent` y antes de la primera barra.

Si un símbolo aparece más de una vez en una producción, se hace necesario distinguir de algún modo de cuál de ellos hablamos cuando hacemos referencia a sus atributos. El siguiente ejemplo nos ayudará a entender el convenio seguido por metacomp. En la producción

```
<A> -> <A> <A> <B> <B>;
```

se asocia a cada instancia del no terminal <A> una variable: A al símbolo de la parte izquierda, A1 al primero de la parte derecha y A2 al segundo de la parte derecha. Las dos instancias del símbolo <B> llevarán asociadas, respectivamente, las variables B1 y B2. Por comodidad, a la variable B1 se accede también con el identificador "B". El sufijo 1 se podrá eliminar siempre que el identificador que se forma con el resto de la cadena no coincida con otro ya existente (coincidencia que sí se da en el ejemplo en el caso de la variable A1, pero no en el de B1). Además, se puede emplear el subrayado para referirse a la última instancia analizada de cada no terminal. Así, A_ en una acción entre la primera y la segunda aparición de <A> se refiere a A1 y entre la segunda y el final de la regla, a A2.

Estas reglas valen incluso en el caso de que haya disyunciones u otros elementos en la parte derecha, es decir, tenemos las mismas variables para la producción

```
<A> -> <A> (<A>)? (<B> | <B>)*;
```

Los no terminales siguen las mismas reglas para su numeración. Así en

```
<A> -> a a
```

se crearán las variables a, a1, a2 y a_. La variable a será un sinónimo de a1 y ambas existirán en las acciones detrás de la primera a; la variable a2 existirá después de la segunda a; y la variable a_ será un sinónimo de a1 entre la primera y segunda a y un sinónimo de a2 después de la segunda a.

5.2. Comunicación con el entorno

El constructor de la clase AnalizadorSintactico tiene un parámetro opcional adicional que permite al entorno transmitir información arbitraria al analizador. El valor que se pase como segundo parámetro está accesible en el atributo mc_entorno del objeto creado. Así, si quisiéramos indicar a un compilador si debe optimizar o no, podríamos hacer lo siguiente. En la llamada a AnalizadorSintactico, indicamos el flag:

```
...
AnalizadorSintactico(f, optimizar)
...
```

Y en las acciones, tomamos la decisión adecuada:

```
<S> -> <Programa>
       ...
       @if self.mc_entorno == "optimizar":@
       @  codigo.optimiza()@
       ...
```

Por defecto, mc_entorno tiene el valor None.

Por otro lado, si queremos devolver información al entorno, podemos utilizar los atributos sintetizados del símbolo inicial de la gramática, que luego está disponible como un atributo del objeto AnalizadorSintactico. Por ejemplo, para devolver resultados podemos hacer:

```
...
<S> -> <Expresion> @S.valor = Expresion.valor@;
...

def main():
  ...
  A = AnalizadorSintactico(L)
  print "El resultado es: ", A.S.valor
  ...
```

5.3. Acciones semánticas del intérprete

Ya estamos en condiciones de escribir las acciones semánticas asociadas al intérprete. Utilizaremos una variable global, tabla, para almacenar los valores de las variables.

```
41 %
42
43 <Entrada> -> @global tabla@
44              @tabla = {}@
45              ( <Sentencia> ";" )+
46              ;
47
48 <Sentencia> -> <Expresion> @print Expresion.v@
49                ( "->" id @tabla[id.lexema] = Expresion.v@ )?
50                ;
51
52 <Expresion> -> <Termino> @Expresion.v = Termino.v@
53                ( op_sum <Termino>
54                  @if op_sum.lexema == "+":@
55                  @  Expresion.v = Expresion.v+Termino2.v@
56                  @else:@
57                  @  Expresion.v = Expresion.v-Termino2.v@
58                )*
59                ;
60

...

74 <Termino> -> <Factor> @Termino.v = Factor.v@
75              ( op_mult <Factor>
76                @if op_mult.lexema == "*":@
77                @  Termino.v = Termino.v*Factor2.v@
78                @else:@
79                @  Termino.v = Termino.v/Factor2.v@
80              )*
81              ;
82
83 <Factor> -> entero @Factor.v = entero.v@
84           |
85              real @Factor.v = real.v@
86           |
87              id @Factor.v = tabla[id.lexema]@
88           |
89              "(" <Expresion> ")" @Factor.v = Expresion.v@
90           ;
91
```

Fíjate en cómo hemos escrito las acciones que ocupan más de una línea. Basta con poner las diversas líneas, encerradas entre arrobas, una a continuación de otra, teniendo cuidado de dejar los espacios necesarios para que el sangrado sea correcto según las normas de Python. De este modo es posible expresar acciones con complejidad arbitraria, aunque es aconsejable, por claridad, utilizar funciones auxiliares definidas en las zonas de código de usuario o en otros ficheros.

Ya hemos codificado nuestro intérprete de expresiones aritméticas. Puede que te preguntes cómo hemos tratado el problema de los tipos, es decir, cómo hemos controlado el comportamiento de la calculadora cuando en la expresión participan números reales. Pues bien, hemos delegado el tratamiento de este problema en el intérprete de Python:

cuando éste ejecuta nuestras acciones semánticas sigue, casualmente, las reglas de promoción de tipos que hemos impuesto a nuestro intérprete de expresiones aritméticas.

5.4. Acciones semánticas del compilador

Para escribir el compilador, no podemos esconder el tratamiento de los tipos. La tabla de símbolos guardará la dirección y tipo de las variables. La variable global `libre` tendrá la primera dirección libre de memoria, que utilizaremos para asignar direcciones a las variables del usuario a medida que las encontremos. El código correspondiente a nuestro compilador podría ser el siguiente:

```
class Variable:
    def __init__(self, dir, tipo):
        self.dir = dir
        self.tipo = tipo

def emite(*instrucciones):
    for instruccion in instrucciones:
        print instruccion
%
<Entrada> -> @global tabla, libre@
             @tabla = {}; libre = 0@
             ( <Sentencia> ";" )+
             ;

<Sentencia> ->
               <Expresion>
               (
                 "->" id
                 @asignacion(Expresion.tipo, id.lexema)@
               )?
               @if Expresion.tipo=="entero":@
               @    emite("PRINT")@
               @else:@
               @    emite("RPRINT")@
               ;

%
def asignacion(tipo, id):
    global libre
    if not tabla.has_key(id):
        tabla[id] = Variable(libre, tipo)
        libre += 1
    r = ""
    if tabla[id].tipo=="real":
        r = "R"
        if tipo=="entero":
            emite("ITOR")
    emite(r+"DUP", r+"POP &%d" % tabla[id].dir)
%
<Expresion> -> <Termino>
               @tipo = Termino.tipo@
               ( op_sum <Termino>
                  @tipo = operacion(op_sum.lexema, tipo,@
                  @                 Termino2.tipo)@
               )*
               @Expresion.tipo = tipo@
               ;

<Termino> -> <Factor>
               @tipo = Factor.tipo@
               ( op_mult <Factor>
                  @tipo = operacion(op_mult.lexema, tipo,@
                  @                 Factor2.tipo)@
               )*
               @Termino.tipo = tipo@
               ;
%
def operacion(op, t1, t2):
    tipo = t1
    r = ""
    if tipo != t2:
        tipo = "real"
        r = "R"
    if t1 != tipo:
        emite("SWAP", "ITOR", "SWAP")
    if t2 != tipo:
        emite("ITOR")
    x = r + {"+":"ADD","-":"SUB","*":"MUL","/":"DIV"}[op]
    emite(x)
    return tipo

%
<Factor> -> entero
              @emite("PUSH %d" % entero.v)@
              @Factor.tipo = "entero"@
            |
            real
              @emite("RPUSH %f" % real.v)@
              @Factor.tipo = "real"@
            |
            id
              @Factor.tipo = lee_var(id.lexema)@
            |
              "(" <Expresion> ")"
              @Factor.tipo = Expresion.tipo@
            ;
%
def lee_var(id):
    if tabla[id].tipo == "entero":
        emite("PUSH &%d" % tabla[id].dir)
    else:
        emite("RPUSH &%d" % tabla[id].dir)
    return tabla[id].tipo

def main():
  AnalizadorSintactico(sys.stdin)
```

Observa cómo hemos intercalado producciones con zonas de código de usuario. Pretendemos de este modo que las definiciones de funciones estén próximas al lugar donde se utilizan. La definición de la clase `Vacia` se encuentra en la primera zona de código del usuario, justo detrás de la especificación léxica.

6. Tratamiento de errores

Los procesadores de lenguaje que hemos aprendido a construir presentan un comportamiento normal ante cadenas correctamente construidas; pero ante entradas incorrectas se detienen indicándonos la presencia de un error que no pueden tratar. Para poder escribir procesadores más flexibles, metacomp permite dotarles de la capacidad de detectar y tratar errores. La detección y tratamiento es diferente según los errores sean léxicos o sintácticos.

6.1. Detección y tratamiento de errores léxicos

Cuando ninguna de las expresiones regulares que hemos proporcionado en la especificación léxica concuerda con prefijo alguno (de talla mayor que cero) del fragmento de fichero que queda por analizar, nos encontramos ante un error léxico. El mecanismo de recuperación de errores léxicos es rígido: consiste en saltarse todos aquellos caracteres que no permiten que haya una concordancia entre una expresión regular y el resto del fichero. Ante un error léxico

se invoca siempre una función denominada `error_lexico`. Esta función recibe dos argumentos: el número de línea en el que se produjo el error y una cadena que contiene todos los caracteres que el procesador ha tenido que saltarse hasta conseguir una nueva concordancia.

La función `error_lexico` no está predefinida: debe definirla el usuario (si no se hace, el programa generado se detendrá indicando que ha encontrado un error léxico que no puede tratar).

Vamos a enriquecer el analizador léxico de nuestros dos procesadores de lenguaje con una función de tratamiento de error. En una zona de código de usuario definimos `error_lexico` de modo que muestre por la salida estándar de error un mensaje informativo del error detectado:

```
20 def error_lexico(linea, cadena):
21   if len(cadena)>1:
22     sys.stderr.write(u"Error en línea %d:" % linea +
23       " La cadena '%s' no se pudo analizar.\n" % cadena)
24   else:
25     sys.stderr.write(u"Error en línea %d:" % linea +
26       " El carácter '%s' no se pudo analizar.\n" % cadena)
27
```

6.2. Detección y tratamiento de errores sintácticos

Los errores sintácticos se tratan en metacomp mediante la pseudo-componente léxica `error`. Esta componente puede aparecer como parte derecha de una producción (a la que llamamos entonces *producción de error*) o como una de las alternativas de una disyunción. Siempre debe estar seguida de acciones semánticas y no puede estar presente en una concatenación.

Las acciones asociadas a `error` son las que se llevarán a cabo cuando se encuentre un error en alguna de las reglas con la misma parte izquierda que la regla de error o en alguna de las otras alternativas de la disyunción si `error` forma parte de una disyunción. Es decir, si en mi gramática añado la regla:

```
<A> -> error @print "ha habido un error"@ ;
```

se escribirá la cadena "ha habido un error" cuando se produzca un error en el análisis del no terminal `<A>` (siempre que el error no se trate en alguno de los niveles inferiores).

Por otro lado, en

```
<A> -> <B>
       ( <C> | <D>
         | error @print "ha habido un error"@
       )
       <E> ;
```

se capturará cualquier error que se produzca mientras se analiza `<C>` o `<D>`, pero no los que se produzcan en `<B>` o `<E>`. Una vez ejecutadas la acciones asociadas a la componente `error`, el análisis prosigue desde ese punto, descartándose la parte del árbol donde se encontró el error. Es decir, podemos interpretar que la parte errónea de la entrada se ha analizado como si fuera la componente `error`.

Puede que sea más fácil entenderlo con un ejemplo. Supongamos que en nuestra gramática añadimos una producción de error para las expresiones:

```
<Expresion> -> error @...tratamiento...@;
```

Veamos ahora qué pasa al analizar una entrada que comience por `2+(3*;`. En el momento en que lee el punto y coma, el analizador está intentando analizar los siguientes no terminales:

⟨Entrada⟩
⟨Sentencia⟩
⟨Expresion⟩
⟨Termino⟩
⟨Factor⟩
⟨Expresion⟩
⟨Termino⟩

Dentro de ⟨Termino⟩, acaba de ver un `op_mult` y se dispone a comprobar si le sigue un ⟨Factor⟩. Dado que el punto y coma no es uno de los primeros de ⟨Factor⟩ y que éste no es anulable, hemos encontrado un error. El analizador repasa la lista anterior de abajo arriba y se encuentra con que ⟨Expresion⟩ puede tratar el error. La ejecución procede entonces por la segunda invocación de ⟨Expresion⟩, abandonando ⟨Termino⟩.

Como habrás imaginado a partir de la descripción anterior, las acciones se implementan mediante el uso de excepciones, por lo que no hace falta analizar explícitamente la pila ni se necesita ningún mecanismo especial para abandonar los no terminales parcialmente analizados.

Teniendo en cuenta este funcionamiento, nuestra producción de error podría ser como la que se muestra en la figura 1. Con la información de la próxima sección podrás interpretar cómo funciona esta regla.

6.2.1. Acciones semánticas en las reglas de error

Las acciones que deben llevarse a cabo al detectarse el error pueden resumirse en:

- Emitir un mensaje de error.
- Avanzar el analizador léxico hasta que lleguemos a una situación que nos permita continuar el análisis.
- Hacer los ajustes necesarios y continuar el análisis.

Mensaje de error Al emitir el mensaje de error se debe procurar que el usuario tenga una idea clara de dónde se encuentra el error y cuál es su causa. Para poder saberlo, metacomp hace que, al comenzar a ejecutarse la acción asociada a un error, la variable `mc_nt` tenga como valor el no terminal que se esperaba analizar en el momento de detectarse el error. Por otro lado, la variable `mc_t` contiene la lista de los terminales que esperaba encontrarse en el momento de producirse el error.

Si queremos saber en qué línea se ha producido el error o qué había en la entrada en ese momento, tenemos que recurrir al analizador léxico. Para acceder a él, se puede utilizar la variable `mc_al`. El método `linea` de esta variable devuelve el número de línea de la componente léxica actual y la componente en sí es el atributo `actual` (su categoría es el atributo `cat` de `mc_al.actual`). Fíjate en cómo hemos hecho para que el mensaje sea amigable con el usuario. Dado que decir algo parecido a "`encontrado token !->`" no

```
<Expresion>  -> error
                  @mensaje = u"Error en línea %d: "% mc_al.linea()@
                  @mensaje += u"expresión incorrecta, he encontrado un %s\n" % nombre[mc_al.actual.cat]@
                  @mensaje += (u"y tenía que haber sido un %s" %@
                  @                  ", un ".join([nombre[n] for n in mc_t[:-1]]))@
                  @if len(mc_t)>1: mensaje += " o un "@
                  @mensaje += nombre[mc_t[-1]]@
                  @mensaje += "\n"@
                  @sys.stderr.write(mensaje.encode("utf-8"))@
                  @mc_al.sincroniza(mc_siguientes["<Expresion>"])@
                  @Expresion.v= None@
                 ;

```

Figura 1: Producción de error para las expresiones

parece presentable, utilizamos el diccionario `nombre` con el contenido siguiente:

```
nombre={
  "op_sum": "operador de suma o resta",
  "op_mult": u"operador de multiplicación o división",
  "id": "identificador",
  "ent": u"número entero",
  "real": u"número real",
  "!;" : "punto y coma",
  "!->" : u"operador de asignación",
  "!(" : u"paréntesis abierto",
  "!)" : u"paréntesis cerrado",
  "mc_EOF" : "fin de la entrada"
}

```

De esta manera, los mensajes son más aceptables.

Sincronización Generalmente, tendremos que realizar una política de sincronización que consista en avanzar el analizador léxico hasta un momento en que se pueda continuar el análisis. Algunas categorías adecuadas para la sincronización son las que están en los conjuntos primeros y siguientes asociados a cada no terminal. Si nos encontramos con uno de los primeros del no terminal, interesará volver a intentar analizarlo. Ésta sería una política adecuada al tratar errores en ⟨Entrada⟩. Por otro lado, si encontramos alguno de los siguientes, merecerá la pena "hacer como si nada", devolviendo el análisis a las producciones superiores. Éste sería el caso en que nos encontramos un error dentro de una expresión. Tanto para una cosa como para la otra, se puede utilizar el método `sincroniza` de `mc_al`. Este método recibe una lista de posibles terminales con los que sincronizarse. La entrada se lee hasta que o bien se encuentra uno de esos terminales o bien se termina el fichero. En el segundo caso, si `mc_EOF` no estaba en la lista pasada como parámetro a `sincroniza`, llama a la función que se le pasa como segundo parámetro (esta función no debe tener ningún parámetro). Este parámetro es opcional y por defecto es `mc_abandonar`, que provoca el abandono del análisis y la salida del constructor del objeto `AnalizadorSintactico`.

Ajustes finales Una vez se ha producido la sincronización, hay que decidir qué acción tomar. Si se va a devolver el control a los niveles superiores, hay que ajustar los valores de los atributos del no terminal para que no se produzcan problemas. En nuestro caso, una posibilidad sería dar un valor `None` a `Expresion.v` y hacer que la acción semántica de ⟨Sentencia⟩ lo tenga en cuenta (no escribir nada si `Expresion.v` es `None`).

Por otro lado, si queremos continuar el análisis en el punto en que hemos tratado el error tenemos que ejecutar la función `mc_reintentar()`. El efecto de esta función es hacer que al terminar la acción de error (es decir, no inmediatamente), se vuelva a intentar analizar el no terminal o la disyunción donde está el tratamiento. Lógicamente, esto tendrá sentido únicamente si hemos encontrado alguno de los primeros[1]; en caso contrario, se producirá un nuevo error.

Hemos comentado antes que el análisis se puede abortar prematuramente al encontrarse `mc_EOF` durante una sincronización. Si queremos tener algo más de control sobre esta situación, debemos definir una función adecuada y pasársela a `mc_al.sincroniza`. Por ejemplo, supongamos que, para presentar los errores de una manera más organizada, vamos guardando los mensajes de error en una lista. Nuestra función podría en primer lugar imprimir los errores de la lista y posteriormente llamar a `mc_abandonar`. También podemos querer abandonar el análisis tras hacer alguna comprobación semántica. En ese caso, podemos llamar a `mc_abandonar` directamente. Ten en cuenta que la llamada a `mc_abandonar` provoca el abandono inmediato del análisis.

Finalmente, en algunos casos puede ser necesario provocar la aparición de un error, por ejemplo si no podemos tratarlo adecuadamente en el nivel en que estamos o si se ha detectado un error sintáctico mediante comprobaciones semánticas. Para ello, podemos utilizar la función `mc_error` a la que se le pasan dos parámetros: el no terminal en que se ha encontrado el error y la categoría o categorías léxicas que se esperaban. El efecto de la llamada es provocar el error de la misma manera que lo hubiera hecho metacomp y dar lugar al mismo tipo de tratamiento.

Tratamiento en nuestro ejemplo Hemos visto antes un posible tratamiento de errores para nuestro ejemplo. Des-

[1] . . . en realidad, también si encontramos alguno de los siguientes y podemos derivar λ, aunque en este caso puede ser mejor seguir.

graciadamente, la regla de error no captura todos los errores (prueba a introducir dos puntos y coma seguidos); puedes pensar cómo harías para que el tratamiento de error fuera más completo (ten en cuenta que el tratamiento para este no terminal sí que está completo). En particular, si existe una producción de error para el símbolo inicial de la gramática, siempre[2] puedes asegurar que capturarás todos los errores (aunque no siempre es el mejor sitio para tratarlos). Para nuestro caso, es interesante que el tratamiento en el símbolo inicial consista en sincronizarse con el punto y coma y después reiniciar el análisis (tras haber avanzado el analizador léxico).

6.2.2. Errores al no encontrar el terminal esperado

En principio, el esquema explicado en el punto anterior debería valer para tratar todo tipo de error. Sin embargo, hay un caso muy frecuente en el que puede ser muy farragoso emplearlo: cuando se espera un terminal de una determinada categoría léxica y se encuentra uno de otra.

Para indicar a metacomp qué acciones queremos que tome en este caso, debemos poner delante de un terminal las acciones que deseemos entre símbolos "$". Igual que en el caso de las acciones semánticas, podemos poner tantas líneas como deseemos, pero es obligatorio que después de la última aparezca un terminal.

Como antes, las acciones deberán emitir un mensaje adecuado, sincronizarse y hacer los ajustes necesarios. Para emitir el mensaje no tenemos problema, ya que sabemos exactamente el terminal y el no terminal implicados sin necesidad de variables auxiliares. En cuanto a la sincronización, podemos hacerla tanto con el propio símbolo como con alguno de sus siguientes. Una advertencia acerca del funcionamiento de metacomp es necesaria aquí. Para metacomp el terminal y la correspondiente acción de error son dos alternativas excluyentes: o encuentra el terminal y almacena sus atributos o no lo encuentra y ejecuta la acción correspondiente. Esto implica que, de ser necesarios los atributos asociados al terminal, han de ser almacenados explícitamente. La manera recomendada consiste en utilizar la asignación `t = mc_al.actual` si `t` es el terminal, nos hemos sincronizado con él y nos interesan sus atributos.

Supongamos que queremos hacer que olvidarse de cerrar un paréntesis dé un aviso, pero que no aborte la evaluación. Podemos cambiar nuestro intérprete de modo que la correspondiente producción de ⟨Factor⟩ tenga la forma:

```
<Factor> -> ...
          "(" <Expresion>
           $sys.stderr.write("Aviso: paréntesis sin cerrar.\n")$
          ")" @Factor.v = Expresion.v@
          ;
```

Date cuenta de que no hemos necesitado sincronizarnos en este caso, ya que hemos asumido que el paréntesis no se ha escrito y no hay atributos de interés.

[2] Bueno, casi siempre, lee la sección 6.2.3.

6.2.3. Errores al no encontrar el fin de fichero

Aunque hayamos dicho antes que todos los errores se pueden capturar asociando una producción de error al símbolo inicial de la gramática, esto no es cierto. Existe un error que no se captura aquí, el que se produce si a una sentencia del lenguaje generado por el símbolo inicial de la gramática no le sigue el fin de fichero.

Para especificar qué hacer en este caso, hay que introducir antes de la primera regla de la gramática el correspondiente tratamiento, en la forma de acciones entre símbolos `$`.

6.3. Información acerca de los no terminales

Para facilitar la escritura de las rutinas de tratamiento de error, metacomp ofrece los diccionarios `mc_primeros`, `mc_siguientes`, `mc_aceptables` y `mc_anulables`. El valor de `mc_primeros["<A>"]` es una lista con los primeros de ⟨A⟩, donde ⟨A⟩ es un no terminal de la gramática. Esta lista no incluye la cadena vacía. Por otro lado, `mc_siguientes["<A>"]` es una lista con los siguientes de ⟨A⟩. Si entre ellos está el fin de fichero, la lista incluirá la categoría `mc_EOF`. En `mc_aceptables["<A>"]` se almacenan los primeros de ⟨A⟩, junto con sus siguientes si es anulable. Finalmente, `mc_anulables["<A>"]` es `True` si ⟨A⟩ es anulable y `False` si no lo es.

7. Miscelánea

7.1. Gramáticas no RLL(1)... pero que metacomp acepta

metacomp sólo rechaza aquellas gramáticas que presentan recursividad por la izquierda. Para el resto de gramáticas, sean o no RLL(1), metacomp siempre produce un analizador a su salida. Eso sí, metacomp nos advertirá de que la gramática de entrada presenta conflictos en la tabla de análisis predictivo. Dependiendo del tipo de conflicto, metacomp usará una estrategia u otra, avisando de la decisión tomada.

Por ejemplo, el (¡clásico!) problema de conflictividad LL(1) que plantea un diseño directo de la sentencia `if-then-else` se resuelve en nuestro caso con la siguiente producción:

```
<S>  -> IF <E> THEN <S> ( ELSE <S> )? ;
```

Al procesar con metacomp una gramática que contenga esta producción, aparecerá por pantalla un mensaje similar a este:

```
Aviso: hay un conflicto en la parte opcional de la línea 6:
   ( ELSE <S> )?
  con el símbolo ELSE. En ese caso, desplazaré.
```

En el apéndice A, puedes encontrar una descripción detallada de las reglas que sigue metacomp para resolver los conflictos.

7.2. Listados de información

Podemos mostrar diversos listados interesantes mediante la opción "`-m`" (o "`--muestra`"). A esta opción le pasamos una lista de parámetros separados por comas que indican qué queremos mostrar:

NUEVO ($\beta 4$)

- `anulables` muestra los componentes de la gramática que son anulables.
- `esquema` muestra el esquema de traducción, esto es, la gramática y las acciones semánticas.
- `gramatica` muestra las reglas de la gramática.
- `lexico` muestra la especificación léxica, incluyendo las categorías inmediatas.
- `primeros` muestra los primeros de los componentes de la gramática.
- `siguientes` muestra los siguientes de los componentes de la gramática.

En los listados de `anulables`, `primeros` y `siguientes` se muestran los componentes correspondientes a los no terminales, las clausuras y las partes opcionales.

Los listados correspondientes a esta opción se muestran por la salida de error estándar.

También están las opciones "`-g`" (o "`--gramatica`"), que es equivalente a "`-m gramatica`", y "`-e`" (o "`--esquema`"), que es equivalente a "`-m esquema`".

7.3. Ayudas para la depuración

Durante el diseño de la gramática es probable que cometamos errores. El procesador generado por metacomp puede enriquecerse con instrucciones que permiten obtener trazas del análisis sintáctico. Si invocamos metacomp con la opción "`-t`" (o "`--traza`"), obtenemos un procesador de lenguaje que nos indica qué producciones va utilizando durante el análisis y qué funciones de análisis (asociadas a no terminales) va activado y desactivando. Esta información la escribe en la salida de error estándar.

Otra ayuda que ofrece metacomp es la posibilidad de obtener el árbol de análisis de la cadena de entrada. Para ello hay que invocar metacomp con la opción "`-A`" (o "`--arbol`"). Los procesadores creados con esta opción escriben en la salida de error estándar una representación del árbol de análisis en el formato aceptado por verArbol.

7.4. Prueba del analizador léxico

Si queremos probar el analizador léxico, podemos emplear la opción "`-S`" (o "`--sololexico`"). El analizador sólo contendrá el código correspondiente al analizador léxico. Además, si no se ha especificado función `main`, se generará una función que lee la entrada estándar y muestra por la salida estándar los componentes que va encontrando.

Hay que tener en cuenta que, si alguna de nuestras categorías se define directamente en la gramática, tendremos que tener, al menos, una regla donde aparezca.

Al usar esta opción, el fichero puede contener simplemente la especificación léxica seguida de una línea que comience por el símbolo de porcentaje (`%`)[3].

7.5. Prueba de la gramática

Si únicamente estamos interesados en probar la gramática, pero no las acciones semánticas asociadas, podemos utilizar la opción "`-p`" (o "`--puro`") que crea analizadores sintácticos "puros". La utilización de estos analizadores es similar a la de los generados normalmente. Si tenemos la entrada en `l`, podemos hacer:

```
A = AnalizadorSintactico(l)
```

A diferencia del caso habitual, durante el análisis no se ejecutarán ni las acciones de la gramática ni las reglas de error. El resultado del análisis se reflejará en el atributo `mc_error` de `A`. Si tiene un valor falso, es que la entrada era correcta. En caso contrario, hay un error en la entrada. La información del error se encuentra en los atributos:

- `mc_lineaError`: número de línea donde se produjo el error.
- `mc_ntError`: no terminal que se estaba analizando en el momento del error.
- `mc_tError`: terminal que ha provocado el error.
- `mc_esperado`: lista de terminales esperados en el momento del error.

Si el error ha sido causado por entrada presente donde se esperaba el final de la entrada, el valor de `mc_ntError` será `None`.

En caso de que no se haya escrito una función `main`, se genera una que analiza la entrada estándar y escribe un mensaje informando de si esta se ha analizado correctamente o de qué error se ha encontrado en caso contrario.

7.6. Restricciones a los terminales y no terminales

Por cada símbolo de una producción, metacomp crea una variable para almacenar sus atributos. Por ejemplo, la producción

```
<S> -> if <E> then <S>
```

haría que metacomp crease las variables `S`, `if1` (también accesible como `if`), `E1` (también accesible como `E`), `then1` (también accesible como `then`) y `S1`. Pero los identificadores `if` y `then` son problemáticos: corresponden a palabras reservadas de Python y, por tanto, no pueden utilizarse como identificadores de variable. metacomp detecta este problema y lo señala como error. No hay problema, sin embargo, con esta otra producción

```
<S> -> If <E> Then <S>
```

[3] El símbolo de porcentaje se utiliza para decidir dónde termina la especificación léxica.

pues, al ser significativa la diferencia entre mayúsculas y minúsculas, ni `If` ni `Then` entran en conflicto con la lista de palabras reservadas de Python.

Tampoco hay problema con

```
<S> -> "if" <E> "then" <S>
```

porque las categorías definidas directamente no generan variable alguna.

A. Manual de referencia de metacomp

En este apéndice presentamos un borrador de lo que podría ser un manual de referencia de metacomp. Comenzaremos explicando los componentes léxicos de los programas metacomp y su sintaxis. Después comentaremos algunas de las características de los programas Python generados por metacomp. Terminaremos comentando las opciones que acepta metacomp.

A.1. Componentes léxicos

Los programas metacomp están formados por componentes léxicos de las siguientes categorías:

- Espacio en blanco.
- Comentario.
- Especificación léxica.
- Símbolo no terminal.
- Símbolo terminal.
- Acción.
- Acción de error directo.
- Código Python.
- Símbolo de error.
- Flecha.
- Punto y coma.
- Asterisco.
- Cruz.
- Interrogante.
- Barra.
- Paréntesis abierto.
- Paréntesis cerrado.
- Fin de fichero.

Espacios en blanco Sirven para separar otros componentes. No son significativos excepto cuando forman parte de otros componentes. Son espacios en blanco los caracteres `␣\n\r\t`.

Comentario Los comentarios comienzan por el carácter "`#`" y terminan con el final de la línea. No son significativos.

Especificación léxica Este es un componente especial que sólo puede aparecer al comienzo del fichero. Está formado por líneas con tres campos. El primer campo es el nombre de una categoría léxica: está formado por letras y el símbolo subrayado (`_`) y no puede ser ni la palabra `error`, ni una palabra reservada de Python (excepto `None`), ni comenzar por la secuencia `mc_`, ni terminar en subrayado. El segundo campo es el nombre de una función Python o la palabra reservada `None`. El tercer campo es una expresión regular según la sintaxis del apéndice A.2. Los campos están separados entre sí por secuencias de espacios y tabuladores.

Las líneas de la especificación se pueden separar con líneas en blanco o líneas de comentario. Estas últimas son líneas en las que el primer carácter no blanco es el carácter `#`. Estas líneas no son consideradas por el analizador léxico.

La especificación termina en el primer fin de línea que preceda a un carácter porcentaje, que no formará parte de la especificación.

Su abreviatura en la gramática es **especificación_léxica**.

Símbolo no terminal Los símbolos no terminales son secuencias de letras y subrayados no terminadas en subrayado y encerradas ente un carácter "menor que" (`<`) y un carácter "mayor que" (`>`).

Su abreviatura en la gramática es **noterminal**.

Símbolo terminal Los símbolos terminales tienen dos formas. Si se refieren a categorías declaradas en la especificación léxica, deben seguir las reglas allí descritas. Para categorías con un único lexema, se puede utilizar la "especificación directa", consistente en escribir el lexema entre comillas dobles (`"`). La cadena encerrada entre comillas se lee "tal cual", en particular, no se interpretan las secuencias de escape y no puede contener comillas dobles. La aparición en el código fuente de la cadena "α" tiene un efecto análogo al que tendría la línea:

`!`α `None` β

si estuviera permitido utilizar la admiración en las categorías léxicas y β fuese una expresión regular cuyo lenguaje asociado fuera $\{\alpha\}$. Esta línea aparecería al principio de la especificación léxica, antes de las líneas puestas por el usuario. No se define un orden para su aparición, pero tampoco hay posibilidad de conflicto.

Su abreviatura en la gramática es **terminal**.

Acción Las acciones que pueden aparecer en la parte derecha de las reglas se especifican encerrándolas entre símbolos arroba (`@`). Las acciones no pueden contener otras arrobas. Entre dos arrobas no puede aparecer un fin de línea. Si la acción necesita varias líneas se pueden concatenar encerrando cada una de ellas entre las correspondientes arrobas. Para el sangrado en el código generado,

se respetan los espacios en blanco detrás de las arrobas izquierdas.

Su abreviatura en la gramática es **acción**.

Acción de error directo Siguen las mismas normas que las acciones de error, pero van encerradas ente símbolos de dólar (`$`).

Su abreviatura en la gramática es **errordos**.

Código Python Son secciones del programa metacomp que comienzan por un símbolo de porcentaje (`%`) y terminan en el siguiente porcentaje que comience una línea (sin blancos precediéndolo) o en el final del fichero. El código encerrado se escribe sin cambios en el fichero generado por metacomp.

Su abreviatura en la gramática es **código**.

Categorías con un sólo lexema Las siguientes categorías tienen un sólo lexema:

Categoría	Abreviatura	Lexema
Símbolo de error	**tokenerror**	`error`
Flecha	**flecha**	`->`
Punto y coma	**pyc**	`;`
Asterisco	**asterisco**	`*`
Cruz	**cruz**	`+`
Interrogante	**interrogante**	`?`
Barra	**barra**	`|`
Paréntesis abierto	**abre**	`(`
Paréntesis cerrado	**cierra**	`)`
Fin de fichero		$_{eof}$

A.2. Sintaxis de las expresiones regulares en metacomp

Las expresiones regulares que acepta metacomp se forman a partir de los literales de cadena vacía, de carácter y de clases de caracteres, combinándolos con los operadores regulares de disyunción, concatenación, clausura, clausura positiva y opcionalidad.

Literal de cadena vacía La cadena vacía se representa en metacomp mediante el carácter λ (0x03BB en Unicode) o mediante un par de paréntesis sin nada entre ellos, `()`.

Literales de carácter En metacomp se puede utilizar cualquier literal de carácter permitido en utf-8 para representar la cadena de un carácter formada por ese carácter, salvo para los caracteres siguientes, que deben escaparse: `|`, `*`, `+`, `[`, `.` (punto), `?`, `(`, `)`, λ, `\`. Además, las secuencias `\n` y `\t` se interpretan como el carácter fin de línea y tabulador, respectivamente.

Literales de clase de caracteres En metacomp se pueden utilizar clases de caracteres para simplificar la escritura de las expresiones. Las clases de caracteres tienen dos formas. La más simple es la formada por el carácter punto y representa cualquier carácter. El resto de clases se forman encerrando entre corchetes un cuerpo que indica qué caracteres forman parte de la clase o están excluidos de ella (lo que sucede si el primer carácter es un circunflejo). En el cuerpo de la clase, los caracteres se representan a sí mismos, con la excepción del guión, el corchete cerrado y el circunflejo. El guión se utiliza para indicar rangos de caracteres. El circunflejo, cuando aparece en la primera posición, indica que la clase representa el complementario de la que representaría si no estuviera. Para incluir un guión en una clase de caracteres, se debe colocarlo en la primera o en la última posición.

Operadores regulares Los operadores regulares que acepta metacomp son, de menor a mayor prioridad:

- Disyunción: es binario, asociativo por la derecha y se representa mediante el carácter barra (`|`). Representa el operador de unión de lenguajes.
- Concatenación: es binario, asociativo por la derecha y está implícito por la yuxtaposición de sus operandos.
- Clausura, clausura positiva y opcionalidad: son unarios y postfijos. La clausura se representa mediante `*`, la clausura positiva mediante `+` y el operador de opcionalidad mediante `?`.

Además, se pueden emplear paréntesis para modificar el orden de evaluación de las expresiones.

A.3. Gramática de los programas metacomp

Los programas metacomp deben seguir la gramática de la figura 2.

A.4. Analizadores léxicos generados por metacomp

Para procesar el fichero de entrada, metacomp crea la clase `AnalizadorLexico`. Los métodos que tiene la clase son:

- `__init__(self, entrada)`: constructor de la clase. El parámetro `entrada` es un fichero abierto para lectura o una cadena.
- `linea(self)`: devuelve el número de línea actual.
- `sincroniza(self, sincr, enEOF = mc_abandonar)`: avanza el análisis léxico hasta encontrar algún componente cuya categoría esté en la lista `sincr` o el final del fichero. Si se alcanza el fin de fichero y `mc_eof` no está en `sincr`, se llama a la función `enEOF`.
- `avanza(self)`: avanza el análisis hasta encontrar un componente léxico o el final del fichero. Devuelve el componente encontrado.

Además, el analizador léxico tiene el atributo `actual`, que contiene el último componente léxico encontrado.

Los componentes léxicos pertenecen a la categoría `ComponenteLexico`, que tiene dos métodos:

⟨Compilador⟩ → **especificación_léxica código**⟨tratEOF⟩⟨Lineas⟩
⟨tratEOF⟩ → (**errordos**)*
⟨Lineas⟩ → (**noterminal flecha**⟨ParteDerecha⟩**pyc**|**código**)*
⟨ParteDerecha⟩ → ⟨Alternativa⟩(**barra**⟨Alternativa⟩)*
⟨Alternativa⟩ → **tokenerror acción**(**acción**)*
⟨Alternativa⟩ → ⟨Elemental⟩(⟨Elemental⟩)*
⟨Alternativa⟩ → λ
⟨Elemental⟩ → **noterminal**
⟨Elemental⟩ → (**errordos**)***terminal**
⟨Elemental⟩ → **acción**
⟨Elemental⟩ → **abre**⟨ParteDerecha⟩**cierra**(**asterisco**|**cruz**|**interrogante**|λ)

Figura 2: Gramática de los programas metacomp

- `__init__(self, cat, lexema, nlinea)`: constructor. Los parámetros son la categoría léxica, el lexema y el número de línea. Se limita a almacenarlos en los correspondientes atributos.

- `__str__(self)`: devuelve una representación de la categoría.

Una limitación importante de los analizadores generados por metacomp es que leen su entrada, cuando es un fichero, utilizando `readlines`. Esto implica que la entrada debe estar disponible por completo antes de empezar el análisis.

Si se desea escribir un analizador léxico distinto del generado por metacomp, se puede escribir un módulo que implemente las clases arriba descritas. Después se compila el programa metacomp utilizando la opción `-l` o `--lexico` seguida del nombre del módulo. El código generado por metacomp no tendrá analizador léxico; en su lugar estará la línea

```
from analex import AnalizadorLexico
```

con `analex` sustituido por el nombre del módulo.

A.5. Algunas características de los analizadores sintácticos generados por metacomp

El código generado por metacomp define la clase `AnalizadorSintactico`, que es la que se encarga de llevar a cabo el análisis sintáctico. El constructor tiene dos parámetros, el segundo de ellos con valor por defecto `None`. El primer parámetro es un fichero abierto para lectura o una cadena. Con este parámetro, crea una instancia de la clase `AnalizadorLexico` y comienza el análisis. El segundo parámetro permite pasar información del entorno al analizador. El constructor se limita a asignarlo a `self.mc_entorno`.

Además del constructor, el analizador tiene un método por cada no terminal de la gramática. Estos métodos, así como los atributos, comienzan por la cadena `mc_`, de modo que el código de usuario puede añadir a `self` los atributos que necesite sin riesgo de colisión. El objeto con los atributos del símbolo inicial que se pasa al correspondiente método de análisis es, a su vez, un atributo del objeto `AnalizadorSintactico` y tiene como nombre el nombre del símbolo inicial. De esta manera, el analizador puede devolver resultados al entorno asignándolos como atributos al símbolo inicial de la gramática.

Dado que el código de las acciones de usuario se encuentra dentro de estos métodos, para escribir en variables globales, es necesario utilizar la declaración `global`.

Por el modo en que se generan los métodos de análisis, se establecen criterios para la resolución de los conflictos RLL(1) de la gramática. En particular:

- Si distintas partes derechas son aceptables para un no terminal ante un terminal dado, se elegirá la primera en aparecer en el código fuente.

- Si un terminal está entre los primeros y los siguientes de una parte opcional de una regla (operador `?`), se asumirá que la parte no se reescribe como la cadena vacía.

- Si un terminal está entre los primeros de dos opciones de una disyunción, se elegirá la primera.

- Si un terminal está entre los primeros y los siguientes de una clausura, se optará por no reescribirla como la cadena vacía.

A.6. Módulos importados

El fichero generado por metacomp importa el módulo `sys`, por lo que las acciones y secciones de código de usuario pueden utilizar sus servicios sin necesidad de importarlo.

A.7. Opciones de la línea de órdenes

Las opciones que acepta metacomp en la línea de órdenes son:

- `-h|--help|-a|--ayuda`: muestra la ayuda y sale.

NUEVO (β3)

- `-s|--salida` *fich*: guarda el programa generado en *fich*. Si el entorno lo permite, le da permisos de ejecución.
- `-C|--codEntrada` *cod*: indica que el fichero de entrada usa la codificación *cod*, por defecto `utf-8`.
- `-c|--codSalida` *cod*: hace que la salida del programa generado use la codificación <cod>, por defecto `utf-8`.
- `-A|--arbol`: añade al ejecutable el código que hace que se escriba el árbol de análisis.
- `-t|--traza`: añade al ejecutable el código que hace que se escriba la traza del análisis.
- `-e|--esquema`: muestra el esquema de traducción, ocultando las zonas de código de usuario, equivale a `-m esquema`.
- `-g|--gramatica`: muestra el esquema de traducción, ocultando acciones, tratamiento de errores y zonas de código de usuario, equivale a `-m gramatica`.
- `-l|--lexico` *fichero*: no genera analizador léxico y en su lugar pone la línea `from` *`fichero`* `import AnalizadorLexico`.
- `-L|--licencia`: muestra la licencia del programa.

NUEVO (β4)

- `-m|--muestra` *qué*: muestra las estructuras indicadas en *qué*, una lista separada por comas de uno o más de las siguientes: `anulables`, `esquema`, `gramatica`, `lexico`, `primeros` y `siguientes`.
- `-p|--puro`: genera un analizador sintáctico puro.
- `-S|--sololexico`: genera únicamente el analizador léxico.

B. Preguntas y respuestas

Pregunta 1: En mi lenguaje hay gran cantidad de palabras clave y quisiera utilizar una tabla en lugar de dar una expresión regular para cada una. ¿Cómo puedo hacerlo?

R: Podemos utilizar la rutina de tratamiento de los identificadores. En la especificación léxica hacemos:

```
id    trata_id    [a-zA-Z_][a-zA-Z0-9_]*
```

La función `trata_id` será la encargada de consultar la lista. Si el lexema corresponde a una palabra clave, cambiará el atributo `cat` del componente que se está tratando:

```
def trata_id(componente):
  if componente.lexema in reservadas:
    componente.cat = componente.lexema
```

Ahora basta con que `reservadas` sea un conjunto con las palabras adecuadas:

```
reservadas = set(["si", ..., "mientras" ])
```

Puede suceder que el conjunto de palabras reservadas del lenguaje no sea disjunto con el de Python:

```
reservadas = set(["if", "then", ..., "while" ])
```

En este caso, podemos pasar las categorías a mayúsculas:

```
def trata_id(componente):
  if componente.lexema in reservadas:
    componente.cat = componente.lexema.upper()
...
<Sentencia> -> IF <Expresion> THEN <Sentencia> ;
```

Pregunta 2: ¿Cómo puedo hacer en la función de tratamiento que se omita un componente léxico?

R: Cambia el valor del atributo `cat` a `None`.

Pregunta 3: A veces mi analizador abandona si está sincronizándose y encuentra el final de fichero. Otras veces sí que puedo tratar bien el error. ¿Qué pasa?

R: La función `mc_al.sincroniza` llama a su segundo parámetro si se encuentra el final del fichero *y éste no estaba en la lista de categorías con las que sincronizarse*. Para tener mejor control de la situación, pasa una función adecuada como segundo parámetro.

Pregunta 4: ¿Cómo puedo pasar información del entorno al analizador y viceversa?

R: Para pasar información del entorno al analizador, puedes emplear el parámetro opcional del constructor de `AnalizadorSintactico`. El valor que le pases estará disponible en el analizador a través de `self.mc_entorno`. Para pasar información desde el analizador al entorno, puedes utilizar los atributos sintetizados del símbolo inicial. Lee la sección 5.2.

Pregunta 5: ¿Cómo puedo crear intérpretes interactivos con metacomp?

R: Si no hay componentes que abarquen más de una línea, se pueden ir leyendo líneas de la entrada y creando con ellas nuevas instancias de la clase `AnalizadorSintactico`. Por ejemplo, el siguiente código corresponde a una sencilla calculadora interactiva:

```
# Blancos:
None    None            [ \t\n]+

# Números:
num     copia_entero    [0-9]+
num     copia_real      [0-9]*\.[0-9]+

%

def copia_entero(componente):
  componente.v = int(componente.lexema)

def copia_real(componente):
  componente.v = float(componente.lexema)
%

<S> -> (<E> @print E.v@
           @global anterior; anterior=E.v@)?;
```

```
<E> -> <T> @E.v = T.v@ ( "+" <T> @E.v = E.v+T2.v@
                       | "-" <T> @E.v = E.v-T2.v@ )* ;
<T> -> <F> @T.v = F.v@ ( "*" <F> @T.v = T.v*F2.v@
                       | "/" <F> @T.v = T.v/F2.v@ )* ;

<F> -> num @F.v= num.v@
     | "-" <F> @F.v = -F1.v@
     | "(" <E> ")" @F.v = E.v@
     | "()" @global anterior; F.v = anterior@ ;

%

def main():
  global anterior
  anterior = 0
  l = sys.stdin.readline()
  while l:
    A = AnalizadorSintactico(l)
    l = sys.stdin.readline()
```

Se utiliza `()` para recuperar el valor de la expresión anterior. No hemos incluido ningún tratamiento de error.