pelele: curso12-13:Práctica 1 (12-13)

Práctica 1: un traductor descendente implementado a mano

Procesadores de Lenguaje, curso 2012-2013

Se trata de diseñar en Java un traductor descendente recursivo según la especificación dada en este enunciado. Los conocimientos necesarios para poder realizar esta práctica se irán viendo en las clases de teoría durante los meses de septiembre, octubre, noviembre y diciembre de 2012. La práctica puede descomponerse en varios módulos: un analizador léxico, un analizador sintáctico descendente recursivo y un traductor implementado sobre este analizador sintáctico. Estos módulos se podrán diseñar e implementar incrementalmente tan pronto como el tema correspondiente haya sido estudiado en clase; es decir, no es nada recomendable esperar a los días previos a la entrega de la práctica para ponerse con ella, sino, más bien, se debe ir trabajando en cada uno de los módulos según la planificación temporal que se indica más adelante en este enunciado.

Consideraciones generales

1. Antes de comenzar a diseñar la práctica, asegúrate de que has leído detenidamente las normas referentes a las prácticas que se encuentran en las transparencias publicadas en el Campus Virtual. Ten en cuenta que, a partir de este curso, **las prácticas serán individuales**.
2. En la sección de materiales del Campus Virtual podrás encontrar unos cuantos ficheros de prueba junto a un programa que determina la corrección de la salida de tu práctica con ellos.
3. Has de entregar un único fichero, llamado plp1.tgz, que cumpla con las especificaciones de este enunciado.
4. Las fechas de entrega de esta práctica son:

    - 1.ª entrega: jueves, 29 de noviembre de 2012, hasta las 23:59
    - 2.ª entrega: miércoles, 5 de diciembre de 2012, hasta las 23:59

Características del código entregado

El archivo tgz que entregues ha de contener exclusivamente el código fuente (con extensión .java) de todas las clases que hayas implementado para tu traductor. Los ficheros del archivo tgz no pueden estar contenidos dentro de ningún directorio. Además, tus clases no pueden pertenecer a ningún paquete ni importar ninguna librería de Java que no sea estándar. Estos ficheros de clases serán compilados por el sistema de corrección automática de la misma manera que se hace en el script corrige.sh incluido en los programas de prueba, así que asegúrate de que tu práctica se compila y ejecuta adecuadamente con este script. Por último, el punto de entrada ha de estar incluido en una clase de nombre plp1:

```
public class plp1 {
   public static void main(String[] args) {
   ...
   }
}
```

El archivo tgz ha de contener también un fichero denominado **plp1.txt** que incluya el DNI y nombre del autor de la práctica y el número de horas dedicadas en total a la realización de la práctica; todo esto en un formato como el siguiente:

```
AUTOR = 12345678; Pérez Pérez, Juan
HORAS = 16
```

Tu programa ha de funcionar sin problemas con la versión del compilador y de la máquina virtual del Java Development Kit 6 (JDK 6) instalada en los laboratorios. Antes de entregar tu archivo tgz descomprímelo y asegúrate de que los ficheros incluidos son correctos y funcionan adecuadamente.

Introducción

Esta práctica consiste en la realización de un programa en Java que lea un fichero (cuyo nombre se obtendrá del primer argumento pasado al programa) en el lenguaje fuente de la gramática que se presenta más adelante, e imprima por la salida estándar su traducción según se comenta en esta página.

Por ejemplo, ante una entrada (correcta) como:

```
program colores;
 function rojo:integer;
  var a,b,c:integer;
 begin (* rojo *)
   a := 7;
   b := a+3
 end;

 function verde:real;
  function fosfo:integer;
   var j:real;
   function rito:real;
    var k:integer;
   begin (* rito *)
     k := 643;
     j := k / 10
   end;
  begin (* fosfo *)
   j := 34.5
  end;
 var m,n:real;
     o,p:integer;
 begin (* verde *)
   o := (27-20) * 3;
   m := o;
   while o>10 do
    begin
      if 2*o = 22 then
        p := 1
      else
        p := 0
      endif;
      o := o - 1
    end
 end;
var a,b,c:integer;
begin (* colores *)
 a := 77;
 writeln(a+3)
end
```

el programa ha de producir la siguiente salida:

```
// program colores

int rojo_a,rojo_b,rojo_c;
```

```
int rojo() {
 rojo_a =i 7 ;
 rojo_b =i rojo_a +i 3 ;
}
double verde_fosfo_j;
int verde_fosfo_rito_k;
double verde_fosfo_rito()
{
 verde_fosfo_rito_k =i 643 ;
 verde_fosfo_j =r itor(verde_fosfo_rito_k) /r itor(10) ;
}
int verde_fosfo()
{
  verde_fosfo_j =r 34.5 ;
}
double verde_m,verde_n;
int verde_o,verde_p;
double verde() {
 verde_o =i (27 -i 20) *i 3 ;
 verde_m =r itor(verde_o) ;
 while ( verde_o >i 10)
 {
   if ( 2 *i verde_o ==i 22 )
     verde_p =i 1 ;
   else
     verde_p =i 0 ;
   verde_o =i verde_o -i 1 ;
 }
}

int main_a,main_b,main_c;
int main() {
 main_a =i 77 ;
 printf("%d\n",main_a +i 3);
}
```

Como se puede observar en el ejemplo de traducción, hay varios aspectos a considerar:

1. En el lenguaje fuente las funciones pueden tener funciones locales (subprogramas), pero en el lenguaje objeto no se permite. Por tanto, en la traducción los subprogramas deben llevar un prefijo que indique su padre (que a su vez puede llevar un prefijo con el abuelo, bisabuelo, etc.)
2. Las variables declaradas en una función son accesibles para los subprogramas de las funciones, por lo que es necesario en la traducción emitir dichas variables como variables globales, con un prefijo similar al de los subprogramas
3. En las asignaciones y expresiones, el lenguaje fuente permite la sobrecarga de operadores (habitual en muchos lenguajes de programación), es decir, permite usar el mismo operador para sumar dos enteros o para sumar dos reales. Sin embargo, en el lenguaje objeto no se permite la sobrecarga de los operadores (ni la de la asignación), por lo que es necesario usar el operador adecuado (en el caso de la suma se usaría '+i' o '+r'), y realizar las conversiones necesarias de entero a real usando la función 'itor'.
4. En el lenguaje fuente, como en Pascal, el operador / siempre denota la división real, por lo que aunque los operandos sean enteros, el resultado será real. Por ejemplo '1/2' es 0.5 (en C y otros lenguajes sería 0). Para la división entera se debe usar el operador 'div' (por ejemplo, '7 div 2' es 3). Este operador no se puede utilizar si cualquiera de los operandos es de tipo real.

Más adelante, en el apartado específico sobre la traducción, aparecen más aspectos a considerar: ámbitos, errores semánticos, etc.

Cualquier error (ya sea léxico, sintáctico o semántico) detectado en el fichero de entrada debe imprimirse por la salida de error estándar según un formato fijo que será descrito con detalle más adelante.

Especificación léxica

Los distintos componentes léxicos (*tokens*) del lenguaje fuente que han de ser reconocidos por el analizador léxico son los que se especifican en el cuadro siguiente:

Elemento léxico	Expresión regular	Subcadena a mostrar
pari	'('	'('
pard	')'	')'
mulop	'*' | '/' | 'div'	'*' '/' 'div'
addop	'+' | '-'	'+' '-'
relop	'<' | '>' |'<=' | '>=' | '=' | '<>'	'<' '>' '<=' '>=' '=' '<>'
pyc	';'	';'
dosp	':'	':'
coma	','	','
asig	':='	':='
var	'var'	'var'
real	'real'	'real'
integer	'integer'	'integer'
program	'program'	'program'
begin	'begin'	'begin'
end	'end'	'end'
function	'function'	'function'
if	'if'	'if'
then	'then'	'then'
else	'else'	'else'
endif	'endif'	'endif'
while	'while'	'while'
do	'do'	'do'
writeln	'writeln'	'writeln'
nentero	('0'..'9')+	numero entero
id	('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9')*	identificador
nreal	('0'..'9')+'.'('0'..'9')+	numero real

El propósito de la tercera columna del cuadro anterior se comentará más adelante al hablar de los errores sintácticos.

El lenguaje es sensible a minúsculas y mayúsculas; por lo tanto, 'cont' y 'ConT' son dos identificadores, y 'beGin' o 'BEgiN' no son la palabra reservada 'begin', son identificadores.

Actuarán como separadores el espacio en blanco, el tabulador, '\t', el retorno de carro, '\r', el salto de línea, '\n' y los comentarios.

El analizador léxico también debe ignorar los comentarios, que comenzarán por '(*' y se extenderán (posiblemente a lo largo de varias líneas) hasta que aparezca la secuencia '*)'. No se permiten comentarios anidados, es decir, el primer '*)' cierra el primer '(*'.

En los ficheros del lenguaje fuente (los que ha de reconocer tu traductor) sólo se utilizarán caracteres ASCII de un byte (es decir, no se utilizarán, ni siquiera en los comentarios, caracteres acentuados, ni eñes, etc.). No uses tampoco esos caracteres en tu implementación de la práctica para evitar posibles problemas en la compilación debidos a la codificación de caracteres.

Mensajes de error léxico

Tras encontrar el primer error, el programa debe finalizar su ejecución y devolver el control al sistema operativo.

Cuando el analizador léxico encuentre un error, tiene que imprimir por la salida de error estándar la siguiente información:

- El código del error.
- La línea y la columna del inicio del lexema que se está procesando cuando ha ocurrido el error.
- Un mensaje descriptivo (según se indica más abajo) sobre el error, seguido de un salto de línea.

Si aparece un carácter de tabulación, la columna se ha de incrementar en uno. El único carácter que incrementa el número de línea es '\n'.

Los distintos errores de origen léxico que pueden aparecer en esta práctica y sus códigos son los siguientes.

(1) Se detecta un carácter que no pertenece al alfabeto de entrada o que no permite extraer un componente léxico correcto:

Error 1 (*fila*,*columna*): caracter '*car*' incorrecto

donde *car* es el carácter en cuestión, y la fila y la columna son las de inicio de su lexema. Cada mensaje de error va seguido de un salto de línea. Date cuenta de que no se escriben con tilde las palabras de los mensajes de error, aunque así lo exijan las normas de acentuación; esto es para evitar problemas con las codificaciones de caracteres al comprobar la salida de tu práctica con la salida correcta de referencia. Es importante de cara a la corrección automática de tu práctica que respetes los espacios en blanco de los mensajes de error.

Por ejemplo, frente a la entrada

```
12#x
```

se debería emitir el error

```
Error 1 (1,3): caracter '#' incorrecto
```

(2) El fichero de entrada acaba en medio de un comentario; en este caso no debe indicarse ni la línea ni la columna:

Error 2: fin de fichero inesperado

Notas técnicas

- Como el fichero de entrada puede ser arbitrariamente grande, no puedes almacenar todo el contenido del fichero en memoria, sino que debes ir procesándolo sobre la marcha, de carácter en carácter.
- Es muy recomendable que implementes una clase para el analizador léxico, digamos AnalizadorLexico, y otra para representar cada *token*, digamos Token. El método 'public Token siguienteToken()' de la clase AnalizadorLexico será el que devuelva tras cada llamada el siguiente token de la entrada y sus atributos, y será invocado por el analizador sintáctico.
- El analizador léxico se ha de implementar utilizando un diagrama de transiciones (DT), diseñado previamente en papel, y que luego debe implementarse según se explicará en las clases de teoría. En ningún caso debe utilizarse un AFD y el algoritmo para realizar análisis léxico utilizando AFDs, como se hacía en cursos anteriores.
- Utiliza la clase RandomAccessFile para acceder al fichero de entrada, usando el método readByte() o read() para leer caracteres. El método readChar() lee caracteres Unicode y no sirve para leer caracteres ASCII.

Especificación sintáctica

La gramática que describe el lenguaje fuente es la siguiente. Aunque es poco relevante para el cometido de tu programa, desde el punto de vista sintáctico, este lenguaje es un subconjunto de Pascal. Esta gramática sólo puede ser modificada en la forma descrita más adelante.

S → **program id pyc** VSp Bloque
VSp → VSp UnVsp
VSp → UnVsp
UnVsp -> **function id dosp** Tipo **pyc** VSp Bloque **pyc**
UnVsp -> **var** LV
LV -> LV V
LV -> V
V -> **id** LI **dosp** Tipo **pyc**
LI -> **coma id** LI
LI -> ε
Tipo -> **integer**
Tipo -> **real**
Bloque -> **begin** SInstr **end**
SInstr -> SInstr **pyc** Instr
SInstr -> Instr
Instr -> Bloque
Instr -> **id asig** E
Instr -> **if** E **then** Instr **endif**
Instr -> **if** E **then** Instr **else** Instr **endif**
Instr -> **while** E **do** Instr
Instr -> **writeln pari** E **pard**
E -> Expr **relop** Expr
E -> Expr
Expr -> Expr **addop** Term
Expr -> Term
Term -> Term **mulop** Factor
Term -> Factor
Factor -> **id**
Factor -> **nentero**
Factor -> **nreal**
Factor -> **pari** E **pard**

Mensajes de error sintáctico

Al igual que en el caso del analizador léxico, en cuanto el analizador sintáctico encuentre el primer error, ha de imprimir un mensaje con un formato estricto por la salida de error estándar y devolver el control al sistema operativo. Los distintos errores de tipo sintáctico que pueden producirse son:

(3) El componente léxico leído no es sintácticamente correcto:

Error 3 (fila,columna): encontrado 'lexema', esperaba *lista*

donde lexema es el lexema del token incorrecto y *lista* es una lista con los elementos esperados en el contexto del error según la tercera columna del cuadro que presenta los componentes léxicos; los elementos de la lista han de separarse con un espacio en blanco; los elementos han de aparecer en el mismo orden relativo que en dicho cuadro y exactamente con la misma grafía (comillas simples incluidas cuando corresponda). La subcadena a mostrar para el final de fichero es *final de fichero* (conforme aparece escrito, sin comillas), y, cuando sea un elemento esperado, debe aparecer en último lugar en

lista.

Un ejemplo de mensaje de error es:

```
Error 3 (6,8): encontrado 'i', esperaba ':='
```

Otro ejemplo es:

```
Error 3 (9,3): encontrado ';', esperaba '(' numero entero identificador numero real final de fichero
```

donde puede verse cuál es la cadena a mostrar en el caso de que lo esperado sea el final del fichero (esta cadena no se muestra en el cuadro de componentes léxicos).

(4) Si lo que provoca el error sintáctico es la aparición prematura del final del fichero, se ha de imprimir el siguiente mensaje de error:

```
Error 4: encontrado final de fichero, esperaba lista
```

donde *lista* sigue de nuevo las directrices anteriores. Este error es de carácter sintáctico y es distinto del error número 2 del analizador léxico.

Notas técnicas

- Implementa tu analizador sintáctico en forma de analizador descendente recursivo. Para ello tendrás que calcular, en primer lugar, a mano, los conjuntos de predicción de cada regla de la gramática. Si la gramática no resultara ser LL(1), debes transformarla para que lo sea (si sabes que es necesario transformarla antes de calcular los conjuntos de predicción, puedes hacerlo).

Traducción

Como ya se ha comentado en la introducción, el programa debe traducir de un subconjunto de Pascal (con alguna modificación) a un lenguaje parecido a C, teniendo en cuenta los siguientes aspectos (además de los mencionados en el ejemplo de la introducción):

1. En el lenguaje fuente, cada subprograma constituye un nuevo ámbito, y las reglas de ámbitos son similares a las de otros lenguajes: no es posible declarar dos veces un identificador en el mismo ámbito (pero sí en ámbitos diferentes), y en cuanto un ámbito se cierra se deben olvidar las variables declaradas en dicho ámbito.
2. Puesto que en la traducción las variables locales a los subprogramas se convierten en variables globales con un prefijo que depende del subprograma en que fue declarada la variable, y puesto que es posible que la variable se use en otros subprogramas (descendientes del que la declaró), una de las formas más sencillas de generar el prefijo correcto para una variable es almacenar en la tabla de símbolos la traducción completa de la variable, con prefijo y nombre de la variable. Seguramente existen otras soluciones, pero probablemente son más complicadas de implementar.
3. En la traducción de las expresiones, cuando se operan números reales y números enteros se deben generar las conversiones de tipo necesarias y utilizar los operadores específicos para cada tipo de dato, como se muestra en el ejemplo.
4. En el lenguaje fuente, los operadores relacionales (**relop**) producen un valor de tipo booleano que no puede operarse con ningún otro operando, ni asignarse a ninguna variable, ni imprimirse. Sin embargo, las expresiones de las instrucciones **if** y **while** deben ser de tipo booleano.
5. En el lenguaje fuente no es posible asignar una expresión de tipo real a una variable de tipo entero.
6. La traducción de la instrucción **writeln** depende del tipo de la expresión, se debe generar la cadena "%d" si es entera o "%g" si es real.

Mensajes de error semántico

Tu traductor ha de detectar los siguientes errores de tipo semántico:

En todos los casos, la fila y la columna indicarán el principio de la aparición incorrecta del token.

(5) No se permiten dos identificadores con el mismo nombre en el mismo ámbito, independientemente de que sus tipos sean distintos. El error a emitir si se da esta circunstancia será:

Error 5 (fila,columna): 'lexema' ya existe en este ambito

(6) No se permite acceder en las instrucción de asignación y en las expresiones a una variable no declarada:

Error 6 (fila,columna): 'lexema' no ha sido declarado

(7) Los identificadores de las instrucciones tienen que ser variables en el lenguaje fuente. Si corresponden a funciones, el error a emitir será:

Error 7 (fila,columna): 'lexema' no es una variable

(8) No se permite asignar un valor de tipo real a una variable de tipo entero:

Error 8 (fila,columna): 'lexema' debe ser de tipo real

(9) No se puede asignar a una variable real o entera una expresión relacional (booleana) :

Error 9 (fila,columna): el operador ':=' no admite expresiones relacionales

(10) Las expresiones de las instrucciones **if** y **while** deben ser de tipo relacional (booleano):

Error 10 (fila,columna): en la instrucción 'lexema' la expresión debe ser relacional

(11) El operador **div** sólo se puede utilizar cuando ambos operandos son de tipo entero:

Error 11 (fila,columna): los dos operandos de 'div' deben ser enteros

(12) No se pueden imprimir expresiones relacionales (booleanas):

Error 12 (fila,columna): 'writeln' no admite admite expresiones booleanas

Notas técnicas

- Aunque la traducción se ha de ir generando conforme se realiza el análisis sintáctico de la entrada, dicha traducción se ha de imprimir por la salida estándar únicamente cuando haya finalizado con éxito todo el proceso de análisis; es decir, si existe un error de cualquier tipo en el fichero fuente, la salida estándar será nula (no así la salida de error).
- Para detectar si una variable se ha declarado o no, y para poder emitir los oportunos errores semánticos, es necesario que tu traductor gestione una **tabla de símbolos** para cada nuevo ámbito en la que se almacenen sus identificadores. En esta página encontrarás una posible implementación e información de su uso.

Planificación de la práctica

Lo que sigue son algunos consejos sobre cómo planificar paso a paso la realización de la práctica.

Analizador léxico

- A la hora de evaluar tu analizador léxico, analiza cuáles son *todos* los posibles casos que se pueden dar y estudia si tu

programa trabaja correctamente con todos ellos.

- Una forma cómoda de evaluar el analizador léxico (si para implementarlo has seguido los consejos del apartado del analizador léxico) es construir un programa que vaya imprimiendo cada uno de los *tokens* encontrados (o el final del fichero, si procede), así como el lexema asociado y la fila y la columna donde comienza. La estructura del método Main en este caso puede ser similar a la siguiente, donde toString() es un método de la clase Token que devuelve una cadena con los atributos del *token*:

```
public static void main(String[] args) {

    AnalizadorLexico s;
    Token t;
    …
    RandomAccessFile entrada = null;

    try {
        entrada = new RandomAccessFile(args[0],"r");
        s = new AnalizadorLexico(entrada);

        while ((t=s.siguienteToken()).tipo != Token.EOF) {
            System.out.print(t.toString());
        }
    }
    catch (FileNotFoundException e) {
    …
    }
…
}
```

- Tu analizador léxico debería estar acabado hacia mediados de octubre para ir cumpliendo con un desarrollo adecuado de la práctica.

Analizador sintáctico

- El analizador sintáctico necesita que el analizador léxico le vaya suministrando los tokens que vaya encontrando en el fichero de entrada. Por ello, asegúrate de que el analizador léxico funciona correctamente antes de empezar a implementar el analizador sintáctico.
- Antes de escribir una sola línea de código, calcula los conjuntos de predicción de cada una de las reglas de la gramática, comprueba si es LL(1), y transformala si no lo es. Entonces sigue las pautas vistas en clase para implementar el analizador descendente recursivo.
- Usa inicialmente programas sencillos para probar el analizador sintáctico y compara las reglas aplicadas por tu programa con las que obtengas al construir manualmente una derivación por la izquierda; una forma fácil de hacerlo es imprimir desde el código del analizador sintáctico el número de regla de derivación aplicada en cada caso. Después ve complicando los ficheros de entrada, asegurándote de que obligan a que se considere varias veces cada una de las reglas recursivas (y cada regla, en general).
- A la hora de evaluar el analizador sintáctico, analiza cuáles son todos los posibles casos que se pueden dar (correctos e incorrectos) y estudia si tu programa trabaja correctamente con todos ellos (o con una muestra suficientemente representativa de ellos).
- El analizador sintáctico descendente debería estar acabado hacia principios de noviembre.

Traductor

- Una vez que el analizador léxico y el analizador sintáctico funcionen correctamente, diseña en papel el esquema de traducción dirigida por la sintaxis (ETDS) y, sólo entonces, implementa cada una de las acciones semánticas sobre el analizador sintáctico.
- Cuanto más extensa sea la batería de programas de prueba que utilices, más oportunidades tendrás de salir airoso de la entrega de esta práctica.

- El traductor descendente recursivo y, por ende, la práctica completa debería estar acabada para mediados de noviembre. Durante los días restantes hasta la fecha de la entrega, puedes evaluar más exhaustivamente todo el sistema. Las fechas de entrega están condicionadas por los días festivos del mes de diciembre, que coinciden con los días en que hay clase de prácticas de la asignatura.