Procesamiento de Lenguajes (PL)
Curso 2018/2019

Práctica 3: traductor descendente recursivo

Fecha y método de entrega

La práctica debe realizarse de forma individual, y debe entregarse a través del servidor de prácticas del DLSI **antes de las 23:59 del 17 de abril de 2019**. Los ficheros fuente en Java se comprimirán en un fichero llamado "`plp3.tgz`", y no debe haber directorios (ni `package`) en él, solamente los fuentes en Java.

Al servidor de prácticas del DLSI se puede acceder de dos maneras:

- Desde la web del DLSI (http://www.dlsi.ua.es), en el apartado "Entrega de prácticas"
- Desde la URL http://pracdlsi.dlsi.ua.es

Una vez en el servidor, se debe elegir la asignatura PL y seguir las instrucciones.

Traducción

La práctica consiste en implementar un traductor descendente recursivo basado en la práctica 1, que traduzca del lenguaje fuente (similar a Pascal) a un lenguaje parecido al C.

Ejemplo

```
program prueba;

begin
 var a : integer;
     b : array [ 1..7, 0..2 ] of real;
     c : pointer of pointer of integer;
     d : real;
     e : array [ 1..5 ] of array [ 3..4 ] of
           pointer of integer;
 endvar

   d := 7 mOD a / 2 * 2.3;
   write ( d + 2 Div 3 - 4.5 );
   begin
     var c:real; endvar
     c := 256 mod 2
   end;
   begin
     var cc:integer; endvar
     begin
       var d:integer;
           b:real;
       endvar
       B := cc;
       write(6)
     end
   end
end
```

```
int main()
{

  int a;
  float b[7][3];
  int** c;
  float d;
  int* e[5][2];



  d = itor(7 % a) /r itor(2) *r 2.3;
  printf("%f",d +r itor(2 /i 3) -r 4.5);
  {
    float _c;
    _c = itor(256 % 2);
  }
  {
    int _cc;
    {
      int __d;
      float __b;
      __b = itor(_cc);
      printf("%d",6);
    }
  }
}
```

Debes considerar los siguientes aspectos al realizar la práctica:

1. **MUY IMPORTANTE:** diseña el ETDS en papel, con la ayuda de algún árbol o subárbol sintáctico. Solamente cuando lo tengas diseñado puedes empezar a transformar el analizar sintáctico en un traductor, que irá construyendo la traducción mientras va analizando el programa de entrada.
2. La traducción de ejemplo está ligeramente modificada para que se vea mejor, no es importante que tenga exactamente los mismos espacios en blanco y saltos de línea.
3. Las variables declaradas en bloques `begin-end` interiores llevan un prefijo con "`_`" que depende del nivel de anidamiento del bloque en el que se han declarado.[1]
4. En las expresiones aritméticas se pueden mezclar variables, números enteros y reales; cuando aparezca una operación con dos operandos enteros, se generará el operador con el sufijo "`i`" (excepto en el caso del operador "`%`", que no lleva sufijo); cuando uno de los dos operandos (o los dos) sea real, el sufijo será "`r`". Si se opera un entero con un real, el entero se convertirá a real usando "`itor`", como en el ejemplo.
5. En las asignaciones pueden darse tres casos:
   - La variable es real y la expresión es entera: en ese caso, la expresión debe convertirse a real con "`itor`"
   - La variable y la expresión son del mismo tipo: en ese caso no se genera ninguna conversión
   - La variable es de tipo entero y la expresión es real: se debe producir un error semántico de tipos incompatibles, como se describe a continuación.
6. Recuerda que el lenguaje fuente es *case insensitive*, es decir, que `NomBRE` y `nOMbre` son la misma variable. En la traducción generada se emitirá siempre el nombre tal y como aparezca en la declaración.
7. En los bloques `begin-end` es posible declarar nuevas variables con el mismo nombre que en ámbitos exteriores, por lo que tendrás que utilizar una tabla de símbolos con gestión de ámbitos.
8. En las expresiones y en la asignación sólo es posible utilizar variables de tipos simples (entero o real).[2]
9. En el lenguaje fuente, los operadores `div` y `mod` operan solamente con números enteros, y el operador `/` siempre realiza una división real aunque los argumentos sean enteros, es decir, `7/2` es `3.5`.

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 del token más relacionado con el error), y emitir el error lo antes posible:

1. 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 semantico (fila,columna): 'lexema', ya existe en este ambito
   ```

2. No se permite acceder en las instrucciones y en las expresiones a una variable no declarada:

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

3. No se permite asignar un valor de tipo real a una variable de tipo entero:

   ```
   Error semantico (fila,columna): 'lexema', tipos incompatibles entero/real
   ```

   En este caso, el lexema será el del operador de asignación.

4. En expresiones y asignaciones solamente se pueden utilizar variables de tipo simple:

   ```
   Error semantico (fila,columna): 'lexema', debe ser de tipo entero o real
   ```

5. Los operadores `div` y `mod` necesitan que ambos operadores sean enteros, en caso contrario se debe emitir un error semántico (uno para el operador izquierdo, y otro para el derecho):

   ```
   Error semantico (fila,columna): 'lexema', el operando izquierdo debe ser entero
   Error semantico (fila,columna): 'lexema', el operando derecho debe ser entero
   ```

6. En la declaración de *arrays*, el segundo número de un rango debe ser mayor o igual que el primero:[3]

[1] Debes usar atributos heredados para resolver este problema.

[2] Sí, las variables de tipo *array* o puntero no se pueden usar y no sirven para nada, pero hay que traducirlas igual (recuerda que esto es una práctica, nada más).

[3] El lexema, fila y columna deben ser los del segundo número.

```
Error semantico (fila,columna): 'lexema', rango incorrecto
```

Por ejemplo, en esta declaración (suponiendo que `variableRepetida` se ha declarado previamente):

```
variableRepetida: array [ 3 .. 2 ] of patata;
```

se debería dar el error de que `variableRepetida` ya existe en el ámbito, a continuación (después de corregir ese error) el error de rango incorrecto, y finalmente el error sintáctico en `patata`.

Notas técnicas

1. 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).

2. 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 la web de la asignatura se publicarán un par de clases para gestionar la tabla de símbolos.

3. La práctica debe tener varias clases en Java:

   - La clase `plp3`, que tendrá solamente el siguiente programa principal (y los `import` necesarios):

```
class plp3 {
    public static void main(String[] args) {

        if (args.length == 1)
        {
          try {
            RandomAccessFile entrada = new RandomAccessFile(args[0],"r");
            AnalizadorLexico al = new AnalizadorLexico(entrada);
            TraductorDR tdr = new TraductorDR(al);

            String trad = tdr.S(); // simbolo inicial de la gramatica
            tdr.comprobarFinFichero();
            System.out.println(trad);
          }
          catch (FileNotFoundException e) {
            System.out.println("Error, fichero no encontrado: " + args[0]);
          }
        }
        else System.out.println("Error, uso: java plp3 <nomfichero>");
    }
}
```

   - La clase `TraductorDR` (copia adaptada de `AnalizadorSintacticoDR`), que tendrá los métodos/funciones asociados a los no terminales del analizador sintáctico, que deben ser adaptados para que devuelvan una traducción (además de quizá para devolver otros atributos y/o recibir atributos heredados), como se ha explicado en clase de teoría. Si un no terminal tiene que devolver más de un atributo (o tiene atributos heredados), será necesario utilizar otra clase (que debe llamarse `Atributos`) con los atributos que tiene que devolver, de manera que el método asociado al no terminal devuelva un objeto de esta otra clase.
   - Las clases `Token` y `AnalizadorLexico` del analizador léxico, y las clases `Simbolo` y `TablaSimbolos` publicadas en la web.