Procesadores de lenguajes. Práctica 6.

TRATAMIENTO DE ERRORES

Índice

1.1 Sincronización de errores

- En la práctica 4 se implementó un analizador sintáctico descendente recursivo basado en gramáticas LL(1) para comprobar si un fichero de entrada es correcto o no.
- En caso de error, el analizador aborta su ejecución lanzando una excepción y se informa de que el resultado es incorrecto.
- Cuando el fichero contiene varios errores, sólo detecta el primero de ellos.
- En un proceso de compilación, obliga al programador a corregir los errores de uno en uno, sin saber cuantos hay en total.
- Para detectar todos los errores es necesario aplicar técnicas de tratamiento de errores, que permitan al analizador sincronizarse de nuevo tras un error para seguir analizando la entrada.

1.1 Sincronización de errores

- El análisis descendente permite desarrollar técnicas de tratamiento de errores por medio de tokens de sincronismo.
- Se analiza cada símbolo de la gramática considerando que va a ser correcto y en el caso de que se detecte un error sintáctico, avanzar en la cadena de tokens hasta consumir el token que debería estar al final del símbolo o alcanzar el token que corresponde al principio del siguiente símbolo.
- Para controlar los errores, el analizador sintáctico debe incorporar dos campos que lleven el control del número de errores detectados “errorCount” y el mensaje de error a mostrar agrupando todos los errores detectados “errorMsg”.
- Al detectar el error se lanza el método catchError(), que incrementa el número de errores, añade la descripción y avanza hasta alguno de los tokens de sincronismo.

1.1 Sincronización de errores

```
private int errorCount;
private String errorMsg;
public int getErrorCount()
   {
     return this.errorCount;
   }
public String getErrorMsg()
   {
     return this.errorMsg;
   }

private void catchError(Exception ex)
   {
     this.errorCount++;
     this.errorMsg += ex.toString();
   }
```

1.1 Sincronización de errores

```
private void skipTo(int[] left, int[] right)
   {
     boolean flag = false;
     if(prevToken.getKind() == EOF || nextToken.getKind() == EOF) flag
= true;
     for(int i=0; i<left.length; i++) {
       if(prevToken.getKind() == left[i]) flag = true; }
     for(int i=0; i<right.length; i++)  {
       if(nextToken.getKind() == right[i]) flag = true;  }

     while(!flag)  {
       prevToken = nextToken;
       nextToken = lexer.getNextToken();
     if(prevToken.getKind()  ==  EOF  ||  nextToken.getKind()  ==  EOF)
flag = true;
     for(int i=0; i<left.length; i++)
     {
       if(prevToken.getKind() == left[i]) flag = true;
     }
     for(int i=0; i<right.length; i++)
     {
       if(nextToken.getKind() == right[i]) flag = true;
     }
   }
}
```

1.1 Sincronización de errores

- Para incluir el tratamiento de errores en el análisis hay que crear un nuevo método asociado a cada símbolo no terminal de la gramática, a cada símbolo A se le asocia parseA() y tryA().
- El método tryA() envuelve la llamada al método parseA() en un bloque try-catch, si en parseA() se produce error se captura y se sincroniza al analizador.

```
private void tryArgumentDecl()
   {
     int[] lsync = { RPAREN };
     int[] rsync = { LBRACE };
     try
     {
       parseArgumentDecl();
     }
     catch(Exception ex)
     {
       catchError(ex);
       skipTo(lsync,rsync);
     }
   }
```

1.1 Sincronización de errores

- Con respecto a los tokens de sincronismo, una opción sencilla es considerar el conjunto siguientes del símbolo como conjunto right de sincronización.
- Cuando los tokens que aparecen en este conjunto son tokens que también pueden aparecer en el contenido del símbolo la sincronización puede ser errónea.
- Esto provoca errores en cascada, es casi imposible de solucionar.
- El objetivo es desarrollar un tratamiento de errores que facilite el proceso de compilación.

1.2 Especificaciones sintácticas descritas en Java

- Existe un tipo de especificación sintáctica especial que no utiliza el formato de descripción EBNF sino una descripción directa de la función en Java.

JAVACODE método_Java

- Este tipo de descripción se suele usar para la gestión de errores, ya que permite definir métodos skipTo(), métodos para eliminar tokens hasta llegar a un token un determinado tipo que sirve como token de sincronismo.

```
JAVACODE
void skipTo(int kind) {
  Token tok = getToken(0);
  while(tok.kind != EOF && tok.kind != kind) tok = getNextToken();}
```

1.2 Especificaciones sintácticas descritas en Java

- Las funciones definidas por medio de declaraciones JAVACODE pueden ser utilizadas como símbolos no terminales y pueden incluirse en las especificaciones EBNF de otras producciones.
- Con este tipo de definición no es posible calcular el conjunto de predicción por lo que no deben usarse nunca como primer elemento de una definición EBNF, ya que el analizador no sabría si ante un determinado token debe ejecutar ese símbolo o no.
- En la siguiente diapositiva vamos a ver un método skipTo() más elaborado entre dos tokens left y rigth. Sincronismo a la izquierda y a la derecha. Se consumen en ambas direcciones, pero prevalece la derecha sobre la izquierda.

1.2 Especificaciones sintácticas descritas en Java

JAVACODE

```
void skipTo(int[] left, int[] right) {
  Token prev = getToken(0);
  Token next = getToken(1);
  boolean flag = false;
  if(prev.kind == EOF || next.kind == EOF) flag = true;
  for(int i=0; i<left.length; i++) if(prev.kind == left[i]) flag =flag = true;
true;
  for(int  i=0;  i<right.length;  i++)  if(next.kind  ==  right[i])
flag = true;                                                    }


  while(!flag) {
    getNextToken();
    prev = getToken(0);
    next = getToken(1);
    if(prev.kind == EOF || next.kind == EOF) flag = true;
    for(int i=0; i<left.length; i++) if(prev.kind == left[i]) flag
= true;
    for(int  i=0;  i<right.length;  i++)  if(next.kind  ==  right[i])
flag = true;
  }
}
```

1.3 Tratamiento de errores

- La herramienta JavaCC utiliza dos tipos de excepciones TokenMgrError y ParseException.
- El lenguaje Java distingue entre errores (subclases de Error que describen problemas graves de los que no se espera recuperación, y el compilador Java no exige su tratamiento) y excepciones (subclases de la clase Exception que describen fallos que podría ser posible una recuperación, el compilador exige que sean tratadas en el código).
- En JavaCC una especificación léxica correcta debe contemplar todas las posibilidades de la entrada, los fallos léxicos se describen como errores que almacena en objetos de la clase TokenMgrError.
- Si el error no es capturado, el analizador escribe un mensaje describiendo error y posición dónde se ha producido.

1.3 Tratamiento de errores

- Los fallos sintácticos son contemplados como excepciones y almacenados en objetos de la clase ParseException. Si las excepciones no son tratadas, el analizador escribe un mensaje “Encontrado … se esperaba uno de los siguientes…” e informa de la posición en el fichero.
- Todas las especificaciones sintácticas descritas en notación EBNF pueden generar tanto errores léxicos como excepciones sintácticas.
- Una producción “tipo símbolo()” genera un método Java con la cabecera “tipo símbolo() throws ParseException.
- Se puede declarar en la cabecera de las producciones.

1.3 Tratamiento de errores

tipo identificador(lista_de_parámetros) throws Excepcion1, Excepcion2 :

{ código_JAVA }

{

descripción_EBNF

}

- Las excepciones declaradas se añaden a ParseException en la cabecera del método Java asociado a la producción del código del analizador.
- El uso de nuevas excepciones resulta muy útil para describir errores semánticos.

1.3 Tratamiento de errores

- Con bloques try-catch no se limita a detectarlos y emitir informes, sino que permite tratarlos.
- Se añaden a las especificaciones EBNF.

```
void Sentencia() :
{}
{
  try { ( SentenciaIF() | SentenciaWHILE() | SentenciaFOR() ) }
  catch(ParseException e) { informar(e); skipTo(PUNTOYCOMA); }
  catch(TokenMgrError e) { informar(e); skipTo(PUNTOYCOMA); }
}
```