Contenido

4.4 UN ANALIZADOR SINTÁCTICO DESCENDENTE RECURSIVO PARA EL LENGUAJE TINY

En esta sección estudiaremos el analizador sintáctico descendente recursivo completo para el lenguaje TINY que se muestra en el apéndice B. El analizador sintáctico construye un árbol sintáctico como se describió en la sección 3.7 del capítulo anterior y, adicionalmente, imprime una representación del árbol sintáctico para el archivo de listado. El analizador sintáctico utiliza la EBNF tal como se proporciona en la figura 4.9, correspondiente al BNF de la figura 3.6 del capítulo 3.

Figura 4.9
Gramática del lenguaje TINY en EBNF

programa → *secuencia-sent*
secuencia-sent → *sentencia* { **;** *sentencia* }
sentencia → *sent-if* | *sent-repeat* | *sent-assign* | *sent-read* | *sent-write*
sent-if → **if** *exp* **then** *secuencia-sent* [**else** *secuencia-sent*] **end**
sent-repeat → **repeat** *secuencia-sent* **until** *exp*
sent-assign → **identificador** := *exp*
sent-read → **read** **identificador**
sent-write → **write** *exp*
exp → *exp-simple* [*op-comparación exp-simple*]
op-comparación → **<** | **=**
exp-simple → *term* { *opsuma term* }
opsuma → **+** | **-**
term → *factor* { *opmult factor* }
opmult → ***** | **/**
factor → **(** *exp* **)** | **número** | **identificador**

El analizador sintáctico de TINY sigue de cerca el esquema del análisis sintáctico descendente recursivo que se dio en la sección 4.1. El analizador sintáctico se compone de dos archivos de código, **parse.h** y **parse.c**. El archivo **parse.h** (apéndice B, líneas 850-865) es muy simple: consta de una declaración simple

```
TreeNode * parse(void);
```

que define la rutina de análisis sintáctico principal **parse**, la cual devuelve un apuntador al árbol sintáctico construido por el analizador sintáctico. El archivo **parse.c** se proporciona en el apéndice B, líneas 900-1114. Se compone de 11 procedimientos mutuamente recursivos que corresponden directamente a la gramática EBNF de la figura 4.9: uno para *secuencia-sent* (*stmt-sequence*), uno para *sentencia* (*statement*), uno para cada una de las cinco clases diferentes de sentencias y cuatro para los diferentes niveles de precedencia de las expresiones. Los no terminales del operador no se incluyen como procedimientos, pero se reconocen como parte de sus expresiones asociadas. Tampoco hay correspondencia de procedimiento para *programa* (*program*), puesto que un programa sólo es una secuencia de sentencias, de manera que la rutina **parse** (**"análisis sintáctico"**) simplemente llama a (**"secuencia-sent"**) **stmt_sequence**.

El código del analizador sintáctico también incluye una variable estática **token** que mantiene el token de búsqueda hacia delante, un procedimiento **match** que busca un token específico, al que llama **getToken** si lo encuentra y declara un error en caso contrario, y un procedimiento **syntaxError** que imprime un mensaje de error en el archivo del listado. El procedimiento principal **parse** inicializa **token** para el primer token en la entrada, llama a **stmt_sequence** y posteriormente verifica si está el final del archivo fuente antes de regresar al árbol construido por **stmt_sequence**. (Si después de que regresa **stmt_sequence** hay más tokens, significa que existe un error.)

El contenido de cada procedimiento recursivo debería ser relativamente autoexplicativo, con la posible excepción de **stmt_sequence**, el cual se escribió en una forma algo más compleja para mejorar el manejo de errores; esto se explicará en el análisis del manejo de errores que en breve se realizará. Los procedimientos de análisis sintáctico recursivo utilizan tres procedimientos utilitarios, los cuales se reúnen por simplicidad en el archivo **util.c**

(apéndice B, líneas 350-526), con la interfaz **util.h** (apéndice B, líneas 300-335). Estos *procedimientos son*

1. **newStmtNode** (líneas 405-421), el cual toma un parámetro indicando la clase de sentencia, y asigna un nuevo nodo de sentencia de esa clase, devolviendo un apuntador al nodo recién asignado;
2. **newExpNode** (líneas 423-440), el cual toma un parámetro indicando la clase de expresión, y asigna un nuevo nodo de expresión de esa clase, devolviendo un apuntador al nodo recién asignado; y
3. **copyString** (líneas 442-455), el cual toma un parámetro de cadena, asigna espacio suficiente para una copia, y copia la cadena, devolviendo un apuntador a la copia recién asignada.

El procedimiento **copyString** es necesario porque el lenguaje C no asigna de manera automática espacio para cadenas y porque el analizador léxico vuelve a utilizar el mismo espacio para los valores de cadena (o lexemas) de los tokens que reconoce.

También se incluye en **util.c** un procedimiento **printTree** (líneas 473-506) que escribe una versión lineal del árbol sintáctico para el listado, de manera que podamos observar el resultado de un análisis sintáctico. Este procedimiento se llama desde el programa principal, bajo el control de la variable global **traceParse**.

El procedimiento **printTree** funciona al imprimir información del nodo y luego efectuar una sangría para imprimir la información del hijo. El árbol real se puede reconstruir a partir de estas sangrías. **traceParse** imprime el árbol sintáctico del programa TINY de muestra (véase el capítulo 3, figuras 3.8 y 3.9, páginas 137-138) en el archivo del listado como se muestra en la figura 4.10.

Figura 4.10
Exhibición de un árbol sintáctico de TINY mediante el procedimiento `printTree`

```
Read: x
If
  Op: <
    const: 0
    Id: x
  Assign to: fact
    const: 1
  Repeat
    Assign to: fact
      Op: *
        Id: fact
        Id: x
    Assign to: x
      Op: -
        Id: x
        const: 1
    Op: =
      Id: x
      const: 0
Write
  Id: fact
```

4.5 RECUPERACIÓN DE ERRORES EN ANALIZADORES SINTÁCTICOS DESCENDENTES

La respuesta de un analizador sintáctico a los errores de sintaxis a menudo es un factor crítico en la utilidad de un compilador. Un analizador sintáctico debe determinar por lo menos si un programa es sintácticamente correcto o no. Un analizador sintáctico que sólo realiza esta tarea se denomina **reconocedor**, puesto que se limita a reconocer cadenas en el lenguaje libre de contexto generado por la gramática del lenguaje de programación en cuestión. Vale la pena establecer que cualquier analizador sintáctico debe comportarse por lo menos como un reconocedor; es decir, si un programa contiene un error de sintaxis, el analizador sintáctico debe indicar que existe *algún* error y, a la inversa, si un programa no contiene errores de sintaxis, entonces el analizador sintáctico no debe afirmar que existe alguno.

Más allá de este comportamiento mínimo, un analizador sintáctico puede mostrar muchos niveles diferentes de respuestas a los errores. Habitualmente, un analizador sintáctico intentará proporcionar un mensaje de error importante, cuando menos para el primer error que encuentre, y también intentar determinar de manera tan exacta como sea posible la ubicación en la que haya ocurrido el error. Algunos analizadores sintácticos pueden ir tan lejos como para intentar alguna forma de **corrección de errores** (o, para decirlo de manera quizás más apropiada, **reparación de errores**), donde el analizador sintáctico intenta deducir un programa correcto a partir del incorrecto que se le proporciona. Si intenta esto, la mayor parte de las veces se limitará sólo a casos fáciles, como la falta de un signo de puntuación. Existe un grupo de algoritmos que se pueden aplicar para encontrar un programa correcto que en cierto sentido se parezca mucho al proporcionado (habitualmente en términos del número de tokens que deben insertarse, eliminarse o modificarse). Esta **corrección de errores de distancia mínima** es por lo regular muy ineficiente como para aplicarse a cualquier error y, en cualquier caso, la reparación de errores que da como resultado a menudo está muy lejos de lo que el programador pretendía. Por consiguiente, es raro que se vea en analizadores sintácticos reales. A los escritores de compiladores les es muy difícil generar mensajes de error significativos sin intentar hacer corrección de errores.

La mayoría de las técnicas para la recuperación de errores son de propósito específico, ya que se aplican a lenguajes específicos y a algoritmos de análisis sintáctico específico, con muchos casos especiales para situaciones particulares. Los principios generales son difíciles de obtener. Algunas consideraciones importantes que se aplican son las siguientes.

1. Un analizador sintáctico debería intentar determinar que ha ocurrido un error *tan pronto como fuera posible*. Esperar demasiado tiempo antes de la declaración del error significa que la ubicación del error real puede haberse perdido.
2. Después de que se ha presentado un error, el analizador sintáctico debe seleccionar un lugar probable para reanudar el análisis. Un analizador sintáctico siempre debería intentar analizar tanto código como fuera posible, a fin de encontrar tantos errores reales como sea posible durante una traducción simple.
3. Un analizador sintáctico debería intentar evitar el **problema de cascada de errores**, en la cual un error genera una larga secuencia de mensajes de error falsos.
4. Un analizador sintáctico debe evitar bucles infinitos en los errores, en los que se genera una cascada sin fin de mensajes de error sin consumir ninguna entrada.

Algunos de estos objetivos entran en conflicto entre sí, de tal manera que un escritor de compiladores tiene que efectuar "convenios" durante la construcción de un manejador de errores. Por ejemplo, el evitar los problemas de cascada de errores y bucle infinito puede ocasionar que el analizador sintáctico omita algo de la entrada, con lo que compromete el objetivo de procesar tanta información de la entrada como sea posible.

4.5.1 Recuperación de errores en analizadores sintácticos descendentes recursivos

Una forma estándar de recuperación de errores en los analizadores sintácticos descendentes recursivos se denomina **modo de alarma**. El nombre se deriva del hecho que, en situaciones

complejas, el manejador de errores consumirá un número posiblemente grande de tokens en un intento de hallar un lugar para reanudar el análisis sintáctico (en el peor de los casos, incluso puede consumir todo el resto del programa, lo que no es mejor que simplemente salir después del error). Sin embargo, cuando se implementa con cuidado, éste puede ser un método para la recuperación de errores mucho mejor que lo que implica su nombre.[4] Este modo de alarma tiene, además, la ventaja de que virtualmente asegura que el analizador sintáctico no caiga en un bucle infinito durante la recuperación de errores.

El mecanismo básico del modo de alarma es proporcionar a cada procedimiento recursivo un parámetro extra compuesto de un conjunto de **tokens de sincronización**. A medida que se efectúa el análisis sintáctico, los tokens que pueden funcionar como tokens de sincronización se agregan a este conjunto conforme se presenta cada llamada. Si se encuentra un error, el analizador sintáctico **explora hacia delante**, desechando los tokens hasta que ve en la entrada uno del conjunto de tokens de sincronización, en donde se reanuda el análisis sintáctico. Las cascadas de errores se evitan (hasta cierto punto) al no generar nuevos mensajes de error mientras tiene lugar esta exploración adelantada.

Las decisiones importantes que se tienen que tomar en este método de recuperación de errores consisten en determinar qué tokens agregar al conjunto de sincronización en cada punto del análisis sintáctico. Por lo general los conjuntos Siguiente son candidatos importantes para tales tokens de sincronización. Los conjuntos Primero también se pueden utilizar para evitar que el manejador de errores omita tokens importantes que inicien nuevas construcciones principales (como sentencias o expresiones). Los conjuntos Primero también son importantes, puesto que permiten que un analizador sintáctico descendente recursivo detecte pronto los errores en el análisis sintáctico, lo que siempre es útil en cualquier recuperación de errores. Es importante darse cuenta que el modo de alarma funciona mejor cuando el compilador "sabe" cuándo *no* alarmarse. Por ejemplo, los símbolos de puntuación perdidos, tales como los de punto y coma, y las comas, e incluso los paréntesis derechos olvidados, no siempre deberían provocar que un manejador de errores consuma tokens. Naturalmente, debe tenerse cuidado para asegurar que no se presente un bucle infinito.

Ilustraremos la recuperación de errores en modo de alarma esquematizando en pseudocódigo su implementación en la calculadora descendente recursiva de la sección 4.1.2 (véase también la figura 4.1). Además de los procedimientos *match* y *error*, que en esencia permanecen iguales (excepto que *error* ya no sale de inmediato), tenemos dos procedimientos más, *checkinput*, que realiza la verificación temprana de búsqueda hacia delante, y *scanto*, que es el token consumidor en modo de alarma propiamente dicho:

```
procedure scanto ( synchset ) ;
begin
  while not ( token in synchset ∪ { $ }) do
    getToken ;
end scanto ;

procedure checkinput ( firstset, followset ) ;
begin
  if not ( token in firstset ) then
    error ;
    scanto ( firstset ∪ followset ) ;
  end if ;
end;
```

Aquí el signo $ se refiere al fin de la entrada (EOF).

4. Wirth [1976], de hecho, llama al modo de alarma la regla de "no alarma", supuestamente en un intento de mejorar su imagen.

Estos procedimientos se utilizan como sigue en los procedimientos *exp* y *factor* (que ahora tienen un parámetro *synchset*):

```
procedure exp ( synchset ) ;
begin
  checkinput ( { (, number }, synchset ) ;
  if not ( token in synchset ) then
    term ( synchset ) ;
    while token = + or token = - do
      match (token) ;
      term ( synchset ) ;
    end while ;
    checkinput ( synchset, { (, number }) ;
 end if;
end exp ;

procedure factor ( synchset ) ;
begin
 checkinput ( { (, number }, synchset ) ;
 if not ( token in synchset ) then
  case token of
  ( : match( ( ) ;
      exp ( { ) } ) ;
      match( ) ) ;
  number :
      match(number) ;
  else error ;
  end case ;
  checkinput ( synchset, { (, number }) ;
 end if ;
end factor ;
```

Advierta cómo *checkinput* es llamado dos veces en cada procedimiento: una vez para verificar que un token en el conjunto Primero sea el token siguiente en la entrada y una segunda vez para verificar que un token en el conjunto Siguiente (o *synchset*) sea el token siguiente en la salida.

Esta forma de modo de alarma generará errores razonables (mensajes de error útiles que también se pueden agregar como parámetros a *checkinput* y *error*). Por ejemplo, la cadena de entrada `(2+-3)*4-+5` generará exactamente dos mensajes de error (uno para el primer signo de menos y otro para el segundo signo de más).

Observamos que, en general, *synchset* se pasa en las llamadas recursivas, con nuevos tokens de sincronización agregados de manera apropiada. En el caso de *factor*, se hace una excepción después de que se ve un paréntesis izquierdo: se llama a *exp* con paréntesis derecho sólo como su conjunto siguiente (*synchset* se descarta). Esto es típico de la clase de análisis de propósito específico que acompaña a la recuperación de errores en modo de alarma. (Hicimos esto de modo que, por ejemplo, la expresión `(2+*)` no generase un falso mensaje de error para el paréntesis derecho.) Dejamos un análisis del comportamiento de este código, así como su implementación en C, para los ejercicios. Por desgracia, para obtener los mejores mensajes de error y recuperación de errores, virtualmente toda prueba de token se debe examinar por la posibilidad de que una prueba más general, o una prueba más temprana, mejore el comportamiento del error.

4.5.2 Recuperación de errores en analizadores sintácticos LL(1)

La recuperación de errores en modo de alarma se puede implementar en analizadores sintácticos LL(1) de manera semejante a como se implementaron en el análisis sintáctico descendente recursivo. Como el algoritmo es no recursivo, se requiere de una nueva pila para mantener los parámetros *synchset*, y debe planearse una llamada a *checkinput* en el algoritmo antes de cada acción generada por el algoritmo (cuando un no terminal está en la parte superior de la pila).[5] Advierta que la situación de error primario ocurre con un no terminal A en la parte superior de la pila y que el token de entrada actual no esté en Primero(A) (o Siguiente(A), si ε está en Primero(A)). El caso en que un token está en la parte superior de la pila, y no es el mismo que el token de entrada actual, normalmente no ocurre, porque los tokens, en general, sólo se insertan en la pila cuando se ven, de hecho, en la entrada (los métodos de compresión de tabla pueden comprometer esto ligeramente). Dejamos las modificaciones para el algoritmo de análisis sintáctico de la figura 4.2, página 155, para los ejercicios.

Una alternativa para el uso de una pila extra es construir de manera estática los conjuntos de tokens de sincronización directamente en la tabla de análisis sintáctico LL(1), junto con las acciones correspondientes que tomaría *checkinput*. Dado un no terminal A en la parte superior de la pila y un token de entrada que no esté en Primero(A) (o Siguiente(A), si ε está en Primero(A)), existen tres alternativas posibles:

1. Extraer A de la pila.
2. Extraer de manera sucesiva tokens de la entrada hasta que se vea un token para el cual se pueda reiniciar el análisis sintáctico.
3. Insertar un nuevo no terminal en la pila.

Seleccionamos la alternativa 1 si el token de entrada actual es \$ o está en Siguiente(A) y la alternativa 2 si el token de entrada actual no es \$ y no está en Primero(A) $\cup$ Siguiente(A). La opción 3 ocasionalmente es útil en situaciones especiales, pero rara vez es apropiada (analizaremos un caso más adelante). Indicamos la primera acción en la tabla de análisis sintáctico mediante la notación *extraer* y la segunda mediante la notación *explorar*. (Advierta que una acción *extraer* es equivalente a una reducción mediante una producción ε.)

Con estas convenciones la tabla de análisis sintáctico LL(1) (tabla 4.4) se verá como en la tabla 4.9. El comportamiento de un analizador sintáctico LL(1) que utilice esta tabla, dada la cadena `(2+*)`, se muestra en la tabla 4.10. En esa tabla el análisis sintáctico se muestra sólo a partir del primer error (de manera que el prefijo `(2+` ya se ha reconocido con éxito). También utilizamos las abreviaturas E para *exp*, E' para *exp'*, y así sucesivamente. Observe que hay dos movimientos de error adyacentes antes que se reanude con éxito el análisis sintáctico. Podemos arreglar suprimir un mensaje de error en el segundo movimiento de error exigiendo, después del primer error, que el analizador sintáctico haga uno o más movimientos con éxito antes de generar cualquier mensaje de error nuevo. De este modo, se evitarían las cascadas de mensajes de error.

Existe (por lo menos) un problema en este método de recuperación de errores que pide una acción especial. Como muchas acciones de error extraen un no terminal desde la pila, es posible que ésta se vuelva vacía, todavía con alguna entrada para el análisis sintáctico. Un caso simple de esto en el ejemplo que acabamos de dar es cualquier inicio de cadena con un paréntesis derecho: esto causará que E se extraiga de inmediato y la pila se quede vacía con toda la entrada esperando ser consumida.

---

5. Las llamadas a *checkinput* al final de una comparación, como en el código en un recursivo descendente, se pueden planear también por medio de una pila especial de símbolos de manera similar al esquema del cálculo de valores de la sección 4.2.4, página 167.

Una acción que puede tomar el analizador sintáctico en esta situación es insertar el símbolo inicial en la pila y explorar hacia delante en la entrada hasta que se vea un símbolo en el conjunto Primero del símbolo inicial.

Tabla 4.9

Tabla de análisis sintáctico LL(1) (tabla 4.4) con entradas de recuperación de errores

M[N, T]	(	número	)	+	-	*	$
exp	*exp* → *term exp'*	*exp* → *term exp'*	extraer	explorar	explorar	explorar	extraer
exp'	explorar	explorar	*exp'* → ε	*exp'* → *opsuma term exp'*	*exp'* → *opsuma term exp'*	explorar	*exp'* → ε
opsuma	extraer	extraer	explorar	*opsuma* → +	*opsuma* → -	explorar	extraer
term	*term* → *factor term'*	*term* → *factor term'*	extraer	extraer	extraer	explorar	extraer
term'	explorar	explorar	*term'* → ε	*term'* → ε	*term'* → ε	*term'* → *opmult factor term'*	*term'* → ε
opmult	extraer	extraer	explorar	explorar	explorar	*opmult* → *	extraer
factor	*factor* → (*exp*)	*factor* → **número**	extraer	extraer	extraer	extraer	extraer

Tabla 4.10

Movimientos de un analizador sintáctico LL(1) utilizando la tabla 4.9

Pila de análisis sintáctico	Entrada	Acción
$ *E' T'*) *E' T*	*)$	explorar (error)
$ *E' T'*) *E' T*	)$	extraer (error)
$ *E' T'*) *E'*	)$	*E'* → ε
$ *E' T'*)	)$	concordar
$ *E' T'*	$	*T'* → ε
$ *E'*	$	*E'* → ε
$	$	aceptar

4.5.3 Recuperación de errores en el analizador sintáctico de TINY

El manejo de errores del analizador sintáctico de TINY, como se da en el apéndice B, es muy rudimentario: sólo está implementada una forma muy primitiva de recuperación en modo de alarma, sin los conjuntos de sincronización. El procedimiento `match` simplemente declara error, estableciendo cuál token que no esperaba encontró. Además, los procedimientos `statement` y `factor` declaran error cuando no se encuentra una selección correcta. El procedimiento `parse` también declara error si se encuentra un token distinto al fin de archivo después de que termina el análisis sintáctico. El principal mensaje de error que se genera

es "unexpected token" ("token inesperado"), el cual puede no ser muy útil para el usuario. Además, el analizador sintáctico no hace un intento por evitar cascadas de error. Por ejemplo, el programa de muestra con un signo de punto y coma agregado después de la sentencia write

```
...
 5: read x ;
 6: if 0 < x then
 7:    fact := 1;
 8:    repeat
 9:       fact := fact * x;
10:       x := x - 1
11:    until x = 0;
12:    write fact; {<-- ¡SIGNO DE PUNTO Y COMA ERRÓNEO! }
13: end
14:
```

provoca que se generen los siguientes *dos* mensajes de error (cuando únicamente ha ocurrido un error):

```
Syntax error at line 13: unexpected token -> reserved word: end
Syntax error at line 14: unexpected token -> EOF
```

Y el mismo programa con la comparación < eliminada en la segunda línea de código

```
...
 5: read x ;
 6: if 0 x then { <-- ¡SIGNO DE COMPARACIÓN PERDIDO AQUÍ! }
 7:    fact := 1;
 8:    repeat
 9:       fact := fact * x;
10:       x := x - 1
11:    until x = 0;
12:    write fact
13: end
14:
```

ocasiona que se impriman *cuatro* mensajes de error en el listado:

```
Syntax error at line 6: unexpected token -> ID, name = x
Syntax error at line 6: unexpected token -> reserved word: then
Syntax error at line 6: unexpected token -> reserved word: then
Syntax error at line 7: unexpected token -> ID, name = fact
```

Por otra parte, algo del comportamiento del analizador sintáctico de TINY es razonable. Por ejemplo, un signo de punto y coma perdido (más que uno sobrante) generará sólo un mensaje de error, y el analizador sintáctico seguirá construyendo el árbol sintáctico correcto como si el signo de punto y coma hubiera estado allí desde el principio, con lo que está realizando una forma rudimentaria de corrección de error en este único caso. Este comportamiento resulta de dos hechos de la codificación. El primero es que el procedimiento `match` no consume un token, lo que da como resultado un comportamiento que es idéntico al de insertar

un token perdido. El segundo es que el procedimiento **stmt_sequence** se escribió de manera que conecte tanto del árbol sintáctico como sea posible en el caso de un error. En particular, se debe tener cuidado para asegurar que los apuntadores hermanos estén conectados dondequiera que se encuentre un apuntador no nulo (los procedimientos del analizador sintáctico están diseñados para devolver un apuntador de árbol sintáctico nulo si se encuentra un error). También, la manera obvia de escribir el cuerpo de **stmt_sequence** basado en el EBNF

```
statement();
while (token==SEMI)
{ match(SEMI);
  statement();
}
```

se puede escribir con una prueba de bucle más complicada:

```
statement();
while ((token!=ENDFILE) && (token!=END) &&
       (token!=ELSE) && (token!=UNTIL))
{ match(SEMI);
  statement();
}
```

El lector puede advertir que los cuatro tokens en esta prueba negativa comprenden el conjunto Siguiente para *stmt-sequence* (*secuencia-sent*). Esto no es un accidente, ya que una prueba puede buscar un token en el conjunto Primero [como lo hacen los procedimientos para *statement* (*sentencia*) y *factor*], o bien, buscar un token que *no* esté en el conjunto Siguiente. Esto último es particularmente efectivo en la recuperación de errores, puesto que si se pierde un símbolo Primero, el análisis sintáctico se detendría. Dejamos para los ejercicios un esbozo del comportamiento del programa para mostrar que un signo de punto y coma perdido en realidad provocaría que el resto del programa se omitiera si **stmt_sequence** se escribiera en la primera forma dada.

Finalmente, advertimos que el analizador sintáctico también se escribió de una manera tal que no puede caer en un bucle infinito cuando encuentra errores (el lector debería haberse preocupado por esto cuando advirtió que **match** no consume un token no esperado). Esto se debe a que, en una ruta arbitraria a través de los procedimientos de análisis sintáctico, con el tiempo se debe encontrar el caso predeterminado de **statement** o **factor**, y ambos *consumen* un token mientras generan un mensaje de error.