Camacho Zavala Ricardo
Valverde Rojas Gustavo

Análisis de problemas encontrados

1. Análisis del comportamiento inicial (Parser recursivo sin transformación)

Recursión infinita y desbordamiento de pila:

En la gramática 1 (E → E + T | T, T → T * F | F, F → (E) | id), la función E() llama a E() de manera recursiva sin consumir ningún token. Esto ocurre porque la producción E → E + T se traduce en una llamada recursiva inmediata a E(), lo que provoca que la pila de ejecución crezca indefinidamente hasta causar un desbordamiento (stack overflow).

El mismo problema se presenta en la gramática 2 (L → A | F, F → (S), S → SL | L, A → num | id) en la función S(), que intenta S() antes que L(), entrando también en recursión infinita.

Además de la recursión infinita, observamos que el parser no es capaz de analizar correctamente las entradas válidas debido a la falta de factorización y a la presencia de recursión izquierda. Por ejemplo, una entrada como "id + id * id" no puede ser analizada porque el parser se queda atascado en la recursión.

Paso	Función Llamada	Acción	Token	Comentario
1	E()	Llama a E()	"id"	Entra en recursión
2	E()	Llama a E()	"id"	Nueva llamada recursiva
...hasta el desbordamiento de la pila				

El parser recursivo directo con gramáticas que contienen recursión izquierda es inviable porque conduce a recursión infinita. Además, la falta de factorización puede llevar a que el parser elija una producción incorrecta, aunque en este caso el problema principal fue la recursión izquierda.

2. Transformación de la gramática y nuevo parser - módulo B.

Eliminación de recursión izquierda:

Gramática 1 original:

Camacho Zavala Ricardo
Valverde Rojas Gustavo

E → E + T | T

T → T * F | F

F → (E) | id

Transformada a:

E → T E'

E' → + T E' | ε

T → F T'

T' → * F T' | ε

F → (E) | id

Gramática 2 original:

L → A | F

F → (S)

S → SL | L

A → num | id

Transformada a:

L → A | F

F → (S)

S → L S'

S' → L S' | ε

A → num | id

Factorización izquierda:

En este caso, no fue necesaria la factorización porque no había producciones con prefijos comunes que causaran ambigüedad en la predicción.

Camacho Zavala Ricardo
Valverde Rojas Gustavo

Los cambios en la implementación del parser:

- Se reestructuraron las funciones para que cada no terminal corresponda a las nuevas producciones.
- Se introdujeron funciones para los no terminales primados (E', T', S') que manejan la repetición mediante recursión por la derecha (que no causa problemas) o iteración.
- El parser ahora consume al menos un token antes de realizar una llamada recursiva, evitando así la recursión infinita.

Comprobación del funcionamiento correcto:

Se probaron las mismas entradas que antes fallaban:

Para gramática 1: "id + id * id", "(id)", etc. Ahora son reconocidas correctamente.

Para gramática 2: "num", "(id num)", etc. También son reconocidas.

El parser ya no entra en bucles infinitos y el análisis es eficiente.

3. Backtracking manual simple - módulo D.

Comprobación sin backtracking:

Se implementaron parsers recursivos para dos gramáticas adicionales sin backtracking:

Gramática 1: S → a A | a B; A → b; B → c

Gramática 2: E → E | Ea | Fb | F; F → (E) | Bbe | Bba; B → Ba | C | b; C → a | CbC

Sin backtracking

Gramática 1:

Entrada "ac": El parser intenta la primera producción (a A) y espera 'b' después de 'a'. Al encontrar 'c', falla y no intenta la segunda producción (a B). Por lo tanto, rechaza una entrada válida.

Gramática 2:

Entrada "(bbe)": El parser se pierde en la recursión izquierda de E → E y B → Ba, causando recursión infinita.

Backtracking manual

El parser se modificó para que guarde la posición actual en la cadena de tokens antes de probar una producción. Si la producción falla, restaura la posición y prueba la siguiente producción. Si todas las producciones fallan, retorna falso.

Problemas del backtracking manual:

Complejidad de implementación: El código se vuelve más complicado porque hay que guardar y restaurar estados manualmente.

Ineficiencia: Puede probar múltiples rutas fallidas antes de encontrar la correcta, lo que es ineficiente en tiempo.

Recursión infinita en gramáticas recursivas por izquierda: El backtracking no resuelve el problema de recursión izquierda, por lo que en gramáticas como la 2, el parser puede entrar en bucles infinitos.

Análisis de Ambigüedad:

Si la gramática es ambigua, el parser con backtracking puede encontrar más de una derivación para una misma entrada. En este caso, el parser simplemente acepta si al menos una derivación es exitosa, pero no informa de la ambigüedad.

El parser recursivo no maneja la ambigüedad de forma inherente, lo que puede ser un problema si se requiere un único árbol de análisis.

4. Reflexión general.

Ventajas del enfoque recursivo:

Simplicidad: Cuando la gramática es adecuada LL(1), el parser es muy sencillo de implementar y entender.

Mapeo directo: Cada regla de producción se traduce en una función, lo que hace el código muy legible.

Camacho Zavala Ricardo
Valverde Rojas Gustavo

¿Por qué transformar la gramática?

Transformar la gramática convierte la gramática en LL(1), lo que permite un parser predictivo eficiente y sin backtracking. Esto es mucho mejor que el backtracking manual porque:

- Es más eficiente.
- Evita la complejidad de guardar/restaurar estados.
- Es determinista, lo que facilita la generación de mensajes de error y la construcción del árbol sintáctico.

Herramientas modernas:

Herramientas como ANTLR o Yacc/Bison generan parsers automáticamente a partir de gramáticas, resolviendo problemas como:

- Eliminación de recursión izquierda.
- Factorización izquierda.
- Manejo de ambigüedades mediante reglas de prioridad y asociatividad.
- Generación de parsers eficientes.

Estas herramientas permiten centrarse en la gramática y no en los detalles de implementación del parser.

En conclusión, la práctica nos permitió comprender la importancia del diseño de gramáticas para los parsers descendentes recursivos. La recursión izquierda genera un problema que debe eliminarse, y la factorización es necesaria para evitar ambigüedades en la predicción. El backtracking manual es una técnica buena para solucionarlos, pero con costos altos en complejidad y eficiencia, por lo que su uso debe limitarse a casos específicos. Para la mayoría de los casos, transformar la gramática a LL(1) y usar un parser predictivo es la mejor opción. Las herramientas modernas de generación de parsers automatizan estas transformaciones, haciendo viable el análisis sintáctico para gramáticas complejas.