Notación Forma Backus-Naur Extendida (EBNF)

Las construcciones repetitivas y opcionales son muy comunes en las escrituras de los lenguajes de programación. Es por eso por lo que en ocasiones se extiende la notación de Backus-Naur (BNF) con notaciones especiales.

La notación es la BNF extendida o EBNF.

- La repetición se simboliza con llaves {}, como $A \rightarrow \{\alpha\}\beta$ en lugar de la regla recursiva por la izquierda $A \rightarrow A\alpha|\beta$
- La opción se simboliza con corchetes, como $A \rightarrow [\alpha]\beta$ en lugar de $A \rightarrow \alpha\beta|\beta$

Con estas reglas el ejercicio es realizar la gramática Tiny utilizando estas dos formas.

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-comparacion exp-simple]*

op-comparación → **<** | **=**

exp-simple → *term* [*opsuma term]*

opsum → +|-

term → *factor [opmult factor]*

opmult → * | /

factor → *(exp)* | ***numero*** | **identificador**

Analizador sintáctico Descendente Recursivo

La idea es considerar la regla gramatical para un no terminal A como una definición a un procedimiento (posiblemente recursivo) que reconocerá una A.

El lado derecho de la regla gramatical especifica la estructura del código para ese procedimiento.

La secuencia de terminales y no terminales en una selección corresponde a concordancias de la entrada y llamadas a otros procedimientos, respectivamente.

Las selecciones corresponden a las alternativas (*switch* o *if*) dentro del código.

Por ejemplo, considere la siguiente gramática

E→ EST | T

S → +|-

T → TMF | F

M → *

F → (E) | N

Entonces un procedimiento que reconoce un factor F será:

```
If token es lpar then
      Empatar un lpar
      Si se reconoce una expresión E
      Empata un rpar
Else if token es número then
      Empata un número
Else
      Señala un error.
```

En este algoritmo suponemos que existe una función que empata el token actual con su parámetro, avanza en la entrada si tiene éxito y señala un error en caso contrario:

If token es el esperado **then**

Lee nuevo token

Else

Señala error.

Asumiremos que señalar un error es simplemente detener el análisis y mostrar un error en pantalla

Utilizando la EBNF para la selección

Considere la regla gramatical:

D → **if** (E) S | **if** (E) S **else** S

La podemos reescribir en EBNF como

D → **if** (E) S [**else** S]

Esto lo Podemos traducer al algoritmo de renocimiento como

```
Empata un if
Empata un lpar ("(")
Reconoce una expresión E
Empata un rpar (")")
Reconoce una sentencia S
If el token es else then
      Empata un else
      Reconoce una sentencia S
```

No podemos utilizar directamente la regla E → EST | T para reconocer una expresión, ya que implica que se llame a sí mismo el procedimiento, es decir, un ciclo infinito. En lugar de eso reescribirnos esta producción en EBNF como E → T {ST}.

Esto lo podemos traducir al algoritmo como

```
Reconoce un término T
While token no es "+" o un "-" do
      Empata un token (ya sea "+" o "-")
      Reconoce un término T
```

En esta parte eliminamos el no terminal S (ya que solo puede reconocer "+" o "-".

Ejercicio en clase:

Utilice la regla T→TMF|F y conviértala a EBNF y además escriba un algoritmo para reconocer la producción.

T→F{MF}

```
Reconoce un factor F
While token es "*" do
      Empata un "*"
      Reconoce un factor F
```

Construcción del árbol sintáctico de una expresión

El siguiente algoritmo corresponde a la construcción de árbol sintáctico de una expresión E:

```
t ← árbol de un término (T)
while token es un "+" o un "-" do:
      n ← nuevo nodo operador con token
      empata un token
      n.izquierdo ← t
      n.derecho ← árbol de un término (T)
      t ← n
return t
```

Construcción del árbol sintáctico de la producción D → if (E) S [else S]

```
Empata un if
Empata un lpar ("(")
t ← nuevo nodo D
t.exp ← árbol de una expresión (E)
empata un rpar (")")
t.hijo_true ← árbol de una sentencia (S)
if token es else then:
      empata un else
      t.hijo_false ← árbol de una sentencia (S)
else
      t.hijo_false ← nulo
return t
```

Tarea: investigar como eliminar la recursividad por la izquierda general (en dos o más pasos)

Factorización por la izquierda

La factorización por la izquierda se requiere cuando dos o más opciones de reglas gramaticales comparten un prefijo en común

$$A \rightarrow \alpha\beta|\alpha\gamma$$

En este caso la solución es simplemente factorizar la α por la izquierda y reescribir la producción como dos producciones:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta|\gamma$$

Ejercicio en clase:

Factorice la producción

sent-if → ***if*** *exp* ***then*** *secuencia-sent* ***end*** | ***if*** *exp* ***then*** *secuencia-sent* ***else*** *secuencia-sent* ***end***

resultado:

sent-if → ***if*** *exp* ***then*** *secuencia-sent sent_if_cont*

sent_if_cont' → ***end*** | ***else*** *secuencia-sent* ***end***