

# Instrucciones para debuggear el Parser

---

Este documento contiene instrucciones sobre cómo obtener información sobre el Parser generado por BNFC y cómo analizar el proceso entero de Parsing al correr una prueba (tanto del parsing exitoso, como de los fallidos).

Al correr el comando de BNFC, se genera un archivo `ParCPP.y` que contiene la especificación de Happy (herramienta para generar Parsers) de la gramática CPP.

El archivo contiene la versión minimalista del Parser y hay 2 operaciones que podemos hacer para aprovechar dicho archivo.

---

## Obtener toda la información del Parser

---

Para obtener un archivo que contenga toda la información del comportamiento del Parser podemos correr el comando:

```
happy -i ParCPP.y
```

Este comando generará un archivo `ParCPP.info` que contiene la tabla entera del **LALR(k)** parser generado por BNFC y más información (cómo la lista de no terminales, terminales, cantidad de reglas, etc).

El siguiente es un ejemplo de un estado de dicho archivo con sus respectivas partes:

State 101 → Estado

ListArg → Arg ',' . ListArg (rule 37)		Reglas
' ) '	reduce using rule 35	Acciones en base a el próximo token a procesar
' bool '	shift, and enter state 32	
' double '	shift, and enter state 33	
' int '	shift, and enter state 34	
' string '	shift, and enter state 35	
' void '	shift, and enter state 36	
%eof	reduce using rule 35	Go-tos dependiendo del tope del Stack al volver de otro estado.
Arg	goto state 89	
ListArg	goto state 157	
Type	goto state 91	

## Buscar Conflictos dentro del archivo de información

Los conflictos de las reglas de la gramática (*shift-reduce* y *reduce-reduce*) se encuentran en la tabla de estados del archivo `ParCPP.info`. Para encontrarlos, se puede buscar la secuencia `(reduce` dentro del archivo.

En la sección de **Acciones** de los estados en los que encuentren `(reduce` habrán conflictos.

Si la línea por encima del `(reduce` tiene una acción de *shift*, el tipo de conflicto es de *shift-reduce* ya que el parser tiene que elegir entre realizar la acción de shift o la de reduce (cómo discutimos en el teórico, por convención se elige la de shift).

En cambio, si la acción encima del conflicto es del tipo *reduce*, el tipo de conflicto será de *reduce-reduce*.

```
State 8
[ Exp1 -> Integer . (rule 7)
  Exp2 -> Integer . (rule 9) ]
' ) '      reduce using rule 9
           (reduce using rule 7)
'*'        reduce using rule 9
           (reduce using rule 7)
```

*Reglas en conflicto*

*reduce-reduce conflict*

En el ejemplo de arriba se puede ver un conflicto de *reduce-reduce* entre la regla 7 y la 9 debido a que ambas producen el mismo resultado en dicho estado.

## Debuggear el Parser para una entrada específica

Y que tenemos la lista de los estados en el archivo `ParCPP.info`, técnicamente se puede hacer el trace desde el estado inicial hasta cualquier expresión que deseemos parsear. Sin embargo incluso para lenguajes de tamaños pequeños dicha tabla se vuelve enorme y es difícil para una persona seguir rigurosamente los pasos hasta alcanzar una expresión en concreto.

Por ello le podemos pedir a Happy que nos agregue información de *debug* en nuestro Parser para poder así tener una salida del trace de la ejecución del Parser.

Para ello podemos correr los siguientes comandos:

1. `happy -da ParCPP.y` va a agregar la salida de debug a nuestro Parser
2. `ghc ./TestCPP.hs` una vez tengamos un parser distinto es importante actualizar el archivo ejecutable que usamos para probar nuestro parser y lexer con la nueva versión del parser.
3. `./TestCPP archivo.cc` una vez actualizado el archivo de Test con el nuevo parser podemos pasarle un archivo para que lo procese y tendremos no

sólo la salida tradicional, sino también la salida del debug en la consola.

Los siguiente son ejemplos de salidas exitosas y fallidas:

```
$ echo "2+3" | ./TestCalc
state: 0, token: 5, action: shift, enter state 4
state: 4, token: 4, action: reduce (rule 3), goto state 5
state: 5, token: 4, action: reduce (rule 8), goto state 9
state: 9, token: 4, action: reduce (rule 7), goto state 11
state: 11, token: 4, action: reduce (rule 5), goto state 10
state: 10, token: 4, action: shift, enter state 13
state: 13, token: 5, action: shift, enter state 4
state: 4, token: 6, action: reduce (rule 3), goto state 5
state: 5, token: 6, action: reduce (rule 8), goto state 9
state: 9, token: 6, action: reduce (rule 7), goto state 16
state: 16, token: 6, action: reduce (rule 4), goto state 10
state: 10, token: 6, action: accept.

Parse Successful!
[Abstract Syntax]
EAdd (EInt 2) (EInt 3)
[Linearized tree]
2 + 3
```

Salida debug

→ Salida normal

```
$ echo "2++" | ./TestCalc
state: 0, token: 5, action: shift, enter state 4
state: 4, token: 4, action: reduce (rule 3), goto state 5
state: 5, token: 4, action: reduce (rule 8), goto state 9
state: 9, token: 4, action: reduce (rule 7), goto state 11
state: 11, token: 4, action: reduce (rule 5), goto state 10
state: 10, token: 4, action: shift, enter state 13
state: 13, token: 4, action: fail.
state: 13, token: 0, action: fail.

Parse Failed...

Tokens:
[PT (Pn 0 1 1) (TI "2"),PT (Pn 1 1 2) (TS "+" 4),PT (Pn 2 1 3) (TS "+" 4)]
syntax error at line 1 before +
```

Salida debug

## Siguiendo el *trace* del debug

Una vez tenemos el archivo `ParCPP.info` y la salida del debug tenemos toda la información sobre los pasos que ha hecho el Parser. Pero para poder comprender cómo actuaron las distintas reglas de la gramática, podemos combinar ambos para seguir el trace de debug.

La salida del debug nos indica qué estado se está visitando en cada instante y qué acción se tomó en cada momento. Esto lo podemos *seguir* en el archivo de `ParCPP.info` para poder observar qué reglas son las que están enumeradas en la salida del debug.

**Nota:** La única información omitida en la salida del debug es que una vez que un estado termina de efectuarse (realiza la acción correspondiente al siguiente token a procesar), si no se mueve a un nuevo estado, vuelve al estado anterior. Una vez en el estado anterior, la única sección de acciones que se pueden realizar es la de los **go-to**.

Es decir: Si pasamos del estado 0 al 9 y en el 9 ya realizamos una acción de reduce (que cambia lo que hay en el tope del stack), en el trace se vuelve al estado 0 y se evalúan los *go-to* dependiendo de lo que haya quedado en el stack para saber el siguiente estado a visitar.