1. Sintaxis concreta de los lambda terminos extendidos

Una gramática que (creemos) genera los lambda términos (extendidos) correctos es:

$$
\begin{array}{rcl}
\langle term \rangle & ::= & "\lambda" \ \langle idents \rangle \ ". " \ \langle term \rangle | \langle no\_abs \rangle \\
\langle no\_abs \rangle & ::= & \langle left\_ap \rangle \ "\oslash \ \lambda" \ \langle idents \rangle \ ". " \ \langle term \rangle | \ \langle left\_ap \rangle \\
\langle left\_ap \rangle & ::= & \langle left\_ap \rangle \ "\oslash" \langle atom \rangle | \ \langle atom \rangle \\
\langle atom \rangle & ::= & \langle ident \rangle \ | \ "(" \ \langle term \rangle \ ")"
\end{array}
$$

- $\oslash$ es el caracter espacio. Observar la importancia de reconocer ese espacio para distinguir las aplicaciones.
- Esta gramática denota la sintaxis concreta del lambda cálculo extendido. Internamente, cuantro trabajemos con una sintaxis abstracta para hacer un parser, las expresiones que se agregan al lambda cálculo por convención -por ejemplo $\lambda x \ y.z$- deberán ser representadas por el lambda término correcto y no el abreviado -(λx . (λy . z))-.
- Aceptamos variables entre paréntesis aunque no son lambda términos válidos. Podríamos arreglarlo pero no es realmente relevante y complicaría (más) la gramática. En general, aceptamos términos con paréntesis exteriores.

Justificación (informal) de porqué creemos que esta gramática representa exáctamente los lambda términos extendidos.
Deberíamos probar: T es generado por la gramática $\Leftrightarrow$ T es un lambda término (extendido) válido.
$\Rightarrow$) TODO LAMBDA TÉRMINO EXTENDIDO t ES ACEPTADO POR ESTA GRAMÁTICA. Un lambda-término válido es:

1. o bien un átomo (term ->*noabs* ->leftap ->atom) , dondé un átomo es una **variable**o un **término entre paréntesis**. Lo llamamos átomo porque no hay lambdas sueltos que se extiendan hasta el final de la expresión, pues en caso de tener lambdas internos, estarán delimitados por paréntesis.
2. o bien una **abstracción sin paréntesis exteriores**. será aceptado por la primera producción, asumiendo que su alcance es máximo.
3. o bien **una aplicación t=t1 t2 sin paréntesis exteriores**. Aquí debemos tener cuidado de manejar correctamente los lambdas sueltos, para no ligar el término t2 dentro del alcance de una abstracción suelta; y también tenemos que forzar la asociatividad a izquierda.
   Para lograr esto último, pensamos que si t es una secuencia de muchas aplicaciones, t2 es el último término y t1 es la secuencia de todas las otras aplicaciones.
   Observemos que:
   - **(*)** t1 no puede ser una abstracción sin paréntesis. es decir, el t1 no tiene lambdas sueltos pues sino, no sería una aplicación ya que el lambda se extendería hasta el final. de aquí podemos decir que t1 es un átomo o una aplición.
   - **(**)**el t2 no puede ser una aplicación sin paréntesis, ya que asociamos a izquierda la aplicación. luego, el t2 será una abstracción sin paréntesis o un átomo.

   Con estas consideraciones, analizamos todos los casos posibles de t:

   3A) el t2 NO ES una aplicación sin paréntesis. Entonces, el término de la derecha es un átomo. al generar t1 t2 comenzaremos con esta sucesión de pasos: (term ->*noabs* ->$leftap$

->$leftap$ atom) . el t1 será un átomo o varias aplicaciones sin paréntesis. Sabemos que entre todas esas aplicaciones sin paréntesis no hay lambdas sueltos por la observación (*) ; luego podemos pensar a t1 como una lista de (uno o más) átomos; el no terminal $left_ap$ se encarga de generarla y además de forzar la asociatividad a izquierda de las aplicaciones no parentizadas.

Hay una recursión a izquierda que solucionaremos al hacer el parser.

3B) t2 ES una aplicación sin paréntesis. Entonces: A t1, por lo mismo que 3A, lo representaremos como una lista de átomos generada por $left_ap$; luego reconocemos un espacio (porque es una aplicación) y luego un lambda cuyo alcance será máximo (t2).

por lo tanto, nuestra gramática acepta (al menos de una forma) todos los lambda términos.

⇐) ESTA GRAMÁTICA NO GENERA COSAS QUE NO SON LAMBDA TÉRMINOS. Por la forma de las reglas, las secuencias de caracteres aceptadas serán siempre lambda términos extendidos.

Además de generar lambda términos extendidos, es importante es que si genera un lambda término, su árbol de parseo tiene que representar la estructura del término y no otra que tenga la misma secuencia de caracteres. Es decir, no queremos que genere la cadena t=λa.b c como una aplición entre λa.b y c; pues aunque la cadena es exactamente esa; el árbol de parseo genera a t como aplicación en vez de una abstracción. (La demostración =>nos garantiza que todo lambda término podrá ser representado en su estructura correcta por nuestra gramática; pero el problema es que no genere también la misma secuencia de caracteres pero con una estructura incorrecta.)

Esto es muy difícil de probar y nos limitaremos a probar muchas cadenas con el parser ya hecho para verificar que los términos sean correctamente representados.

2. Sintaxis abstracta de los lambda terminos extendidos

** ACÁ VA MÁS O MENOS LO QUE ESTÁ EN EL ARCHIVO AST QUE USÉ PARA PARSING, NO SÉ CUÁNTO CAMBIARA PARA EL PARSER NUEVO. LA IDEA SERÍA Q NO HAYA PARÉNTESIS Y QUE lambda x y . z sea aceptada como lambda x (lambda y . z)